UNIVERSIDAD POLITÉCNICA DE MADRID
FACULTAD DE INFORMÁTICA

Poly-Controlled Partial Evaluation and its Application to Resource-Aware Program Specialization

PhD Thesis

Claudio J. G. Ochoa
March 2007

PhD Thesis

Poly-Controlled Partial Evaluation and its Application to Resource-Aware Program Specialization

presented at the Computer Science School
of the Technical University of Madrid
in partial fulfillment of the degree of
Doctor in Computer Science

PhD Candidate: **Claudio Ochoa**

Licenciado en Ciencias de la Computación
Universidad Nacional de San Luis, Argentina

Master of Computer Science
University of Illinois at Urbana-Champaign, US

Advisor: **Germán Puebla**

Profesor Titular de Universidad

Madrid, March 2007

Abstract

Partial Evaluation is an automatic technique for program optimization. The aim of partial evaluation is to specialize a program with respect to part of its input, which is known as the *static data*. Existing algorithms for on-line partial evaluation of logic programs, given an initial program and a description of run-time queries, deterministically produce a specialized program. The quality of the code generated by partial evaluation of logic programs greatly depends on the *control strategy* used. Unfortunately, the existence of sophisticated control rules which behave (almost) optimally for all programs is still far from reality.

The main contribution of this thesis is the development of *Poly-Controlled Partial Evaluation*, a novel framework for partial evaluation which is *poly-controlled* in that it can take into account *repertoires* of control strategies instead of a single, predetermined combination (as done by traditional partial evaluation). This approach is more flexible than existing ones since it allows assigning *different* control strategies to different call patterns, thus generating results that cannot be obtained using traditional partial evaluation. As a result, sets of candidate specialized programs can be generated, instead of a single one. In order to make the algorithm fully automatic, it requires the use of self-tuning techniques which allow automatically measuring the quality of the different candidate specialized programs. Our approach is *resource aware* in that every solution obtained by poly-controlled partial evaluation is assessed by means of fitness functions, which can consider multiple factors such as run-time and code size for the specialized programs. The framework has been implemented in the `CiaoPP` system, and tested on numerous benchmarks. Experimental results show that our proposal obtains better specializations than those achieved using traditional partial evaluation, especially in the context of resource-aware program specialization.

Another main contribution of this thesis is the presentation of a unifying view

to the problem of eliminating superfluous polyvariance in both partial evaluation and abstract multiple specialization, through the use of a minimization step that collapses equivalent predicate versions. This step can be applied to any Prolog program (that may include builtins or calls to external predicates) being specialized. Additionally, we offer the possibility of collapsing *non strictly equivalent* versions, in order to obtain smaller programs.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Logic Programming and Program Specialization

1.1.1 Declarative Programming Languages

High level languages are characterized by allowing the programmer to write programs not in terms of the particular machine being used, but rather in terms of the tasks the programs must perform. Thus, the programmer does not have to worry about the specifics of the machine. This results in a less time-consuming and error-prone developing process. Programs written in such high level languages are automatically translated into the language of a particular machine by another program referred to as *compiler*.

An important kind of high level languages are the so-called *declarative languages*. They are called declarative in contrast to the traditional high level languages such as C, Pascal, Java, Ada, etc., which are generally referred to as *imperative languages*. The main difference between declarative languages, a good example of them being logic programming in its pure form, and imperative languages, is that in the former the programmer only needs to express *what* the program should compute. In imperative languages it is also required to express *how* to compute it by explicitly specifying in the program the control flow.

Among the most prominent members of declarative languages we can find *logic programming* and *functional programming*. Logic programming is based on

first-order logic and automated theorem proving, while functional programming has its roots in λ-calculus. In both approaches, a program is considered a *theory* while its execution consists in performing *deduction* from such a theory (sometimes complemented by induction or abduction). Also, modern functional logic languages like Curry [51, 52] and Toy [87] combine the most important features of functional *and* logic languages (see [50] for a survey).

1.1.2 Logic Programming

The *Logic Programming* paradigm [65, 66, 24] is characterized by its appropriateness for knowledge representation and has been used for the implementation of expert systems, knowledge bases, etc. Such applications are in general complex and with a strong symbolic component.

Among the most characteristic and useful features of logic programming languages we can mention:

- they can compute with *partially specified data*,
- the input/output characteristics of predicate arguments is not fixed beforehand,
- they allow *non-determinism*, making them well-suited for applications like parsing,
- they provide for *automatic memory management*, thus avoiding a major source of errors present in other programming languages (specially imperative ones).

It is worth to note that late implementations of logic programming languages have become very efficient, and many times they reach and even surpass the speed of imperative languages for some applications.

1.1.3 Program Specialization and Resource-Awareness

Program specialization is an automatic technique for *program optimization*. In logic programming, given a program P and a (possibly partially instantiated) query G for P, the goal of program specialization is to derive a more efficient

program P'_G that gives exactly the same answers for any instance $G\theta$ of G as P does [42]. Among the most well-known program specialization techniques we can mention *partial evaluation*, *program slicing*, as well as other (compiler-based) techniques for optimizing programs in order to perform more efficient computations.

Partial Evaluation *Partial evaluation* [59, 60, 85, 44] is a source-to-source program transformation technique which specializes programs by fixing part of the input of some source program P and then pre-computing those parts of P that only depend on the known part of the input. The so-obtained transformed programs are less general than the original one, but can be much more efficient. The part of the input that is fixed is referred to as the *static* input, while the remainder of the input is called the *dynamic* input.

In general, most partial evaluators are not *resource aware*, as they focus on time-efficiency, the main goal being the generation of specialized programs which run faster than the original. Other factors such as the size of the specialized program (also called *residual program*), and the memory required to run it, are most often neglected, a relevant exception being the works [34, 28, 27].

Program Slicing *Program slicing* [120, 53], originally proposed as a technique for program debugging, has also been proposed as a resource aware program specialization technique for declarative languages [111, 99, 97, 95]. Program slicing is a method for decomposing programs by analyzing their data and control flow. As already mentioned, it was first proposed as a debugging tool to allow a better understanding of the portion of code which revealed an error. Since this concept was originally introduced by Weiser [127, 128]—in the context of imperative programs—it has been successfully applied to a wide variety of software engineering tasks (e.g., program understanding, debugging, testing, differencing, specialization, merging). Although it is not so popular in the declarative programming community, several slicing techniques for declarative programs have also been developed during the last decade (see, e.g., [47, 76, 96, 111, 116, 119, 122, 96, 77, 100]).

1.2 Overview of the Thesis

1.2.1 Thesis Objectives

The final objective of the work presented in this thesis is the development, implementation, and experimental assessment of a *poly-controlled partial evaluator*. Poly-controlled partial evaluation [110] is a powerful *resource aware* approach for program specialization. It takes into account *repertoires* of global control and local control rules instead of a single, predetermined, combination—as done by traditional partial evaluation—. Thus, *different* control strategies can be assigned to different call patterns, obtaining results that are *hybrid* in the sense that they cannot be obtained using a single combination of control rules, as traditional partial evaluation does.

Poly-controlled partial evaluation can be implemented as a *search-based* algorithm, producing *sets* of candidate specialized programs (most of them hybrid), instead of a single one. The quality of each of these programs is assessed through the use of different fitness functions, which can be *resource aware*, taking into account multiple factors such as run-time, memory consumption, and code size of the specialized programs, among others.

In this way, we will try to fill the existing gap of building a resource aware partial evaluator. Poly-controlled partial evaluation will bring along other advantages. Since it can generate hybrid solutions not achievable by traditional partial evaluation, we hope that for some problems and some fitness functions, hybrid solutions will have better performance than *pure* solutions, i.e., solutions obtained by using a single combination of control rules. Also, existing partial evaluators offer a wide set of parameters and flags to be set in order to deal with termination problems, or to obtain better specialized programs. The drawback is that the result of the interaction of such parameters is often very difficult to predict, even for experienced users. We will aim at implementing a partial evaluator that is auto-tunable, in the sense that it will try to automatically set some parameters, making it also more user-friendly, especially for novice users.

1.3 Structure of the Work

This thesis consists of five parts. Each of these parts is described in detail below.

Part I: Technical Background

In order to make this thesis as self-contained as possible, we start by providing some basic knowledge on the terminology of first-order logic, logic programming and partial evaluation.

The roots of logic programming in first-order logic are described in Chapter 2, with special emphasis on the syntax and semantics of logic programs.

Chapter 3 describes the basics of partial evaluation of logic programs, in terms of *SLD semantics*, introduced in Chapter 2. Control issues of partial evaluation are introduced in this chapter, since they play a very important role in the poly-controlled partial evaluation framework.

Part II: Reducing the Size of Specialized Programs

This part deals with the problem of eliminating unneeded polyvariance in partial evaluation. Polyvariant specialization allows the generation of multiple versions of a procedure, which can be separately optimized for different uses. Though polyvariance is often very important for achieving good specialized programs, it also sometimes results in unnecessarily large residual programs. This problem not only affects code size, sometimes specialized programs run slower due to cache miss effects [34, 121].

A possible solution to this problem is to introduce a minimization step which identifies sets of *equivalent* versions, and replace all occurrences of such versions by a single one. Previous work on eliminating superfluous polyvariance has dealt with *pure* logic programs (programs containing no builtins) and a very limited class of builtins. Chapter 4 tackles the problem of performing this minimization step even in the presence of calls to (any) external predicate, including builtins, libraries, other user modules, etc. Also, we propose the possibility of collapsing versions which are not strictly equivalent. This allows trading time for space and can be useful in the context of embedded and pervasive systems.

Note that this minimization step can be applied to specialized programs obtained by either traditional partial evaluation or poly-controlled partial evaluation.

Part III: Poly-Controlled Partial Evaluation: Foundations

In this part we introduce the main idea of poly-controlled partial evaluation.

Chapter 5 explains the dilemma of choosing adequate control rules when specializing programs through partial evaluation. We show by means of simple examples that the existence of sophisticated control rules which behave (almost) optimally for all programs is still far from reality.

As already mentioned, poly-controlled partial evaluation tries to cope with this problem by allowing the use of different control strategies for different call patterns, obtaining potentially different specialized programs depending on the control strategy used for each call pattern.

We formalize poly-controlled partial evaluation and present two algorithms implementing it. The first algorithm is greedy, and uses a *pick* function to non-deterministically select a control strategy to be used at each moment. The second algorithm is *search-based*, producing *sets* of candidate specialized programs (most of them hybrid), instead of a single one. The quality of each of these programs is assessed through the use of different *resource aware* fitness functions. Some preliminary evaluation results are also provided in this chapter.

Then, Chapter 6 studies the properties of the solutions (specialized programs) obtained by poly-controlled partial evaluation. In particular, we are interested in determining the heterogeneity of the solutions. If the solutions are different enough when compared with one another there will be more chances of finding interesting solutions, i.e., it will be more probable that these solutions can be better than any solution obtained by traditional partial evaluation in similar conditions (i.e., using the same control strategies).

Part IV: Poly-Controlled Partial Evaluation In Practice

In this part we explain the difficulties of implementing the algorithms of poly-controlled partial evaluation, and introduce different techniques for dealing with such problems.

Chapter 7 explains in detail, and through a simple experiment, the main problem poly-controlled partial evaluation suffers when implemented as a

search-based algorithm: its search space experiments a (potentially) exponential growth. We identify the causes of the problem in this chapter, whilst solutions to this problem are proposed in the following chapters of this part.

Chapter 8 explores some techniques for pruning the search space of poly-controlled partial evaluation when implemented as a search-based algorithm. The proposed techniques are based on heuristics, they are simple to understand and implement, and in many cases they achieve a drastic reduction of the size of the search space. It is well known that heuristics may behave well in some cases and not so well in others. Since in this context *behaving not so well* would mean pruning away the solutions of maximal fitness, we empirically check whether this is the case by running these heuristics against a good number of benchmark programs.

In Chapter 9 we propose a *branch and bound*-based pruning technique. This technique outperforms the previous one in that it guarantees that solutions of maximal fitness are not lost. The main drawbacks of this technique are the facts that it is more difficult to implement, and that, in order to prune branches, we need to evaluate intermediate configurations, which introduce a non negligible cost.

Finally, we propose in Chapter 10 an *oracle-based* poly-controlled partial evaluation algorithm. This algorithm aims at achieving results comparable to those of the search-based algorithm introduced in Chapter 5, while having a specialization cost that is a constant factor of that of traditional partial evaluation. Basically, given a call pattern, we first apply all control strategies to it and then *an oracle* makes an informed decision—based on heuristics— of which control strategy is the most promising one. Thus, similarly to the greedy algorithm from Chapter 5, the most promising intermediate solution is kept active while the rest are discarded, thus avoiding search, and traversing just one branch of the search tree.

Part V: Poly-Controlled Partial Evaluation: Implementation

In this part we present the current implementation of poly-controlled partial evaluation in `CiaoPP` [55], the pre-processor of `Ciao`.

Chapter 11 shows some guidelines on the use of poly-controlled partial

evaluation in CiaoPP, by means of an example session, where we enumerate the available flags for adjusting the behaviour of the poly-controlled partial evaluator. We also show how this framework is user-friendly for novice users, who just need to set very few parameters through a graphical interface in order to run the poly-controlled partial evaluator, but also allows expert users to tweak several different parameters by setting different flags in a expert-oriented interface.

Then, in Chapter 12 we provide some conclusions of this thesis.

Appendices

Finally, we add some useful appendices.

In Appendix A we describe the fitness functions that are used in order to evaluate the different candidate specialized programs that are found by poly-controlled partial evaluation. As already mentioned, these functions are resource-aware, in that they can take different parameters into account, other than runtime, such as size of the resulting specialized program, memory taken by the programs, etc.

Appendix B describes the set of benchmark programs used throughout this thesis. Some of these benchmarks have been borrowed from Michael Leuschel's *Dozen of Problems of Partial Deduction* library [79], while some others have been adapted from Lam and Kusalik's set of problems [69]. The rest of benchmarks are taken from different sources, such as Prolog libraries, CiaoPP [55] analysis benchmarks, internet, etc.

As explained in Section 1.1.3, program slicing is another *resource-aware* program specialization technique. We have undertaken an implementation of a slicer in CiaoPP, based on the ideas of [77]. In Appendix C we describe a slicing session of a Ciao program in CiaoPP.

1.4 Main Contributions

The main contributions of this thesis are described below. Some of these results have already been published and presented in international forums, in which case the relevant publication(s) is(are) explicitly mentioned. Also, some of these

contributions have been made in collaboration with other researchers in addition to the thesis supervisor. This is also explicitly mentioned below.

- The main contribution of this thesis is the development, implementation, and experimental assessment of the novel concept of *poly-controlled partial evaluation*. The most important advantages of poly-controlled partial evaluation over traditional partial evaluation are:

  - it allows obtaining *better specialized programs* than traditional partial evaluation. Moreover, in most cases these programs cannot be obtained through traditional partial evaluation using the same control strategies.
  - it is a *resource-aware* approach, taking into account factors such as size of the compiled residual program, and the memory required to run the residual program, besides the speed of the residual program.
  - it is *not yet another control strategy*, but a framework allowing the co-existence and cooperation of any set of control strategies. In fact, poly-controlled partial evaluation will benefit from any further research on control strategies.
  - it is *user-friendly*, allowing the user to simultaneously experiment with different combinations of parameters in order to achieve a specialized program with the desired characteristics.
  - it performs *on-line partial evaluation*, and thus it is fully automatic, and it can take advantage of the great body of work available for *on-line* partial evaluation of logic programs.

  This framework has been presented on the 8th ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP'2006) [110].

- We have studied the properties of the different specialized programs generated by the poly-controlled partial evaluation algorithm. A paper on this work has been presented in the 15th Workshop on Functional and (Constraint) Logic Programming (WFLP'2006) [92]. This paper will be published as a special number of Electronic Notes in Theoretical Computer Science (Elsevier) [98].

- Two algorithms for poly-controlled partial evaluation have been implemented:

  - One of them is *search-based*, relatively simple to implement, but suffering from an exponential blowup of its search space. For this reason, in this thesis we also tackle the problem of pruning the search space in different ways, in order to make this algorithm able to deal with realistic Prolog programs. These pruning techniques and the obtained results have been presented in the ACM SIGPLAN 2007 Symposium on Partial Evaluation and Program Manipulation (PEPM'2007) [93].
  - The other one is *greedy*. It is based on a variation of the search-based algorithm, and relies on an *oracle* that decides which control strategy is the most promising for every call pattern. This algorithm is almost as efficient as traditional partial evaluation in terms of specialization time and memory consumption, achieving at the same time specialized programs as good as those obtained by the search-based approach. Details on the implementation of this algorithm, and experimental results have already been submitted to a relevant international conference on the subject.

- The problem of superfluous polyvariance has been studied both in the context of abstract multiple specialization [130, 108] and in the context of partial evaluation of normal logic programs [84]. The common idea is to identify sets of versions which are *equivalent* and replace all occurrences of such versions by a single, canonical, one. In this thesis we compare different approaches for controlling polyvariance, and we also extend previous approaches in two ways:

  - First, we tackle in an accurate way the case in which programs contain *external predicates*, i.e., predicates whose code is not defined in the program being specialized, and thus it is not available to the specializer.
  - Second, previously proposed minimization techniques do not provide any degrees of freedom at the minimization stage. We propose the possibility of collapsing versions which are not *strictly* equivalent. This is

achieved by residualizing certain computations for external predicates which would otherwise be performed at specialization time. This allows automatically trading time for space.

– we present the first experimental evaluation of the benefits of post-minimization in partial evaluation.

Interestingly, this approach can be applied to both traditional partial evaluation and poly-controlled partial evaluation. This work, co-authored with Manuel Hermenegildo, has been published in the 15th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'2005) [94].

Part I

Technical Background

Chapter 2

Logic and Logic Programming

This chapter provides an essential background in first-order logic and logic programming. It is mainly inspired in [80], which, in turn, it is based on [86] and [7], and adheres to the same terminology. If desired, advanced readers can quickly skim through this chapter, or skip it completely.

2.1 Syntax of Logic Programs

We start by briefly introducing the syntax of well-formed formulas of a first order theory.

Definition 2.1.1 (alphabet)**.** *An* alphabet *consists of the following classes of symbols:*

1. *variables*
2. *function symbols*
3. *predicate symbols*
4. *connectives*
5. *quantifiers*
6. *punctuation symbols*

Classes 1 to 3 vary from alphabet to alphabet, while classes 4 to 6 are the same for all alphabets.

Function and predicate symbols have an associated *arity*, a natural number indicating how many arguments they can take. *Constants* are function symbols of arity 0, while *propositions* are predicate symbols of arity 0.

The connectives are negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), implication ($\leftarrow$), and equivalence ($\leftrightarrow$). The quantifiers are the existential quantifier ($\exists$), and the universal quantifier ($\forall$). Finally, the punctuation symbols are "(", ")" and ",". To avoid having formulas cluttered with brackets, we give connectives and quantifiers the following precedence, with the highest precedence at the top:

$$\neg, \forall, \exists$$
$$\vee$$
$$\wedge$$
$$\leftarrow, \leftrightarrow$$

Throughout this thesis, we adhere as much as possible to the following notational conventions:

- Variables will be denoted by uppercase letters—possibly subscripted—, usually taken from the later part from the (Latin) alphabet, such as *X, Y, Z*.
- Constants will be denoted by lowercase letters, usually taken from the beginning of the (Latin) alphabet, such as *a, b, c*, while other function symbols will be denoted by lowercase letters such as *f, g, h*.
- Predicates will be denoted by lowercase letters, such as *p, q, r*.

We give now a series of basic definitions.

Definition 2.1.2 (term)**.** *A* term *is inductively defined as follows:*

- *A variable is a term.*
- *A constant is a term.*
- *A function f of arity $n > 0$ applied to a sequence of terms $t_1, \ldots, t_n$, denoted by $f(t_1, \ldots, t_n)$, is a term.*

Definition 2.1.3 (atom)**.** *An* atom *is defined inductively as follows:*

- *A proposition is an atom.*
- *A predicate p of arity $n > 0$ applied to a sequence of terms $t_1, \ldots, t_n$, denoted by $p(t_1, \ldots, t_n)$, is an atomic formula, or more simply, an* atom*.*

The function *pred* applied to a given atom A, i.e., $pred(A)$, returns the corresponding predicate symbol p/n for A.

Definition 2.1.4 (formula)**.** *A* (well-formed) formula *is defined inductively as follows:*

- *An atom is a formula.*
- *If F and G are formulas, then so are $(\neg F)$, $(F \wedge G)$, $(F \vee G)$, $(F \leftarrow G)$, and $(F \leftrightarrow G)$.*
- *Given a formula F and a variable X, then $(\forall X\ F)$ and $(\exists X\ F)$ are formulas.*

For example, $\forall(p(X, g(X)) \leftarrow q(X) \wedge \neg r(X))$ is a formula, whose informal semantics is "for every X, if $q(X)$ is true and $r(X)$ is false, then $p(X, g(X))$ is true".

Some important classes of formulas, especially in the context of logic programming, are defined below.

Definition 2.1.5 (expression)**.** *An* expression *is either a term, an atom or a conjunction or disjunction of atoms.*

Definition 2.1.6 (clause)**.** *A* clause *is a formula of the form $\forall H_1 \vee \ldots \vee H_m \leftarrow B_1 \wedge \ldots \wedge B_n$, where $m \geq 0$, $n \geq 0$ and $H_1, \ldots, H_m, B_1, \ldots, B_n$ are all atoms. $H_1 \vee \ldots \vee H_m$ is called the* head *of the clause and $B_1 \wedge \ldots \wedge B_n$ is called the* body *of the clause.*

A (normal) program clause *is a clause of the form $H \leftarrow B_1 \wedge \ldots \wedge B_n$, where H is an atom.*

A definite program clause *is a normal program clause where $B_1 \wedge \ldots \wedge B_n$ are atoms.*

A fact *is a program clause of the form $H \leftarrow$.*

A query *or* goal *is a clause of the form $\leftarrow B_1 \wedge \ldots \wedge B_n$, with $n > 0$.*

A definite goal *is a goal where $B_1 \wedge \ldots \wedge B_n$ are atoms.*

It is important to distinguish the scope of variables present in a given formula.

Definition 2.1.7 (scope)**.** *The* scope *of* $\forall X$ *(resp.* $\exists X$*) in a given formula* $(\forall X\ F)$ *(resp.* $(\exists X\ F)$*) is* F*. A* bound occurrence *of a variable* X *inside a formula* F *is any occurrence immediately following a quantifier or an occurrence within the scope of a quantifier* $\forall X$ *or* $\exists X$*. Any other occurrence of a variable is said to be* free*.*

Definition 2.1.8 (closed formula)**.** *A* closed formula *is a formula with no free variables.*

Definition 2.1.9 (universal and existential closure)**.** *Given a formula* F*, the* universal closure *of* F*, denoted by* $\forall(F)$*, is the closed formula obtained by adding a universal quantifier to every free variable in* F*. Similarly, the* existential closure *of* F*, denoted by* $\exists(F)$*, is obtained by adding an existential quantifier to every free variable in* F*.*

Universal quantifiers are usually omitted when writing logic programs, and commas are used instead of conjunctions in the body. For instance, the formula $\forall X \forall Y p(X,Y) \leftarrow q(X) \wedge r(Y)$ is written as $p(X,Y) \leftarrow q(X), r(Y)$. Throughout this thesis we adhere to this convention.

After the definitions above, we can now define two important concepts, that of a *first order language*, and that of a *program.*

Definition 2.1.10 (first order language)**.** *The* first order language *given by an alphabet* A *consists of the set of all formulas constructed from the symbols of* A*.*

Definition 2.1.11 (program)**.** *A* (normal) program *is a set of (normal) program clauses. A* definite program *is a set of definite program clauses.*

2.2 Semantics of Logic Programs

The declarative *semantics* of a program is given by the semantics of formulas in first-order logic, assigning meaning to formulas in the form of *interpretations* over some domain D. This means that every function symbol of arity n is assigned a n-ary mapping $D^n \mapsto D$, each predicate symbol of arity n is assigned a n-ary relation on D (i.e., a subset of D^n), and variables range over D. Finally, each

formula is assigned a truth value (`true` or `false`) depending on the truth value of every subformula. This intuition can be formalized as follows.

Definition 2.2.1 (model)**.** *A* model *of a formula is an interpretation in which the formula has the truth value* `true` *assigned to it.*

For example, let I be an interpretation whose domain D is the set of natural numbers $\mathbb{N}$ with the following mappings

$$a \mapsto 1$$
$$b \mapsto 2$$
$$p \mapsto \{(1)\}$$

Then the truth value of $p(a)$ under I is `true` and the truth value of $p(b)$ is `false`. So I is a model of $p(a)$ but not of $p(b)$.

Definition 2.2.2 (logical consequence)**.** *A formula F is a logical consequence of a set of formulas S, denoted by $S \models F$, if F is assigned the truth value* `true` *in all models of S.*

The following shorthands are used for formulas:

- Given a formula F, then $F \leftarrow$ denotes the formula $F \leftarrow true$ and $\leftarrow F$ denotes the formula $false \leftarrow F$.
- The *empty clause* is a clause of the form $\leftarrow$, and corresponds to the formula $false \leftarrow true$, i.e., a contradiction.

Given a *definite* program P, and since P is just a set of clauses, and clauses are simply formulas, the logical meaning of P might be seen as the set of all formulas F for which $P \models F$ [1]. Thus, from a programming point of view, we are interested in the *bindings* made for all variables of P to obtain each formula in F.

Definition 2.2.3 (substitution, binding)**.** *A* substitution θ *is a finite set of the form $\theta = \{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$, where $\theta(X_i) = t_i$ for all $i = 1, \ldots, n$ (with*

[1] In the case of *normal* programs, since negations can occur in the bodies of clauses, the truth of $\neg F$ can propagate further and may be used to infer positive formulas as well. For a more detailed discussion on this, see [8, 80]. In this work, we consider negations as regular builtins.

$X_i \neq X_j$ *if* $i \neq j$*) and* $\theta(X) = X$ *for any other variable* X*, and where* t_i *are terms (with* $t_i \neq X_i$*).*

Each element $X_i \mapsto t_i \in \theta$ *is called a* binding.

We denote with ϵ the empty substitution. Also, $vars(E)$ denotes the set of variables occurring inside an expression E, and $dom(\theta)$ denotes the set of variables affected by substitution θ, i.e., $dom(\{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}) = \{X_1, \ldots, X_n\}$.

Definition 2.2.4 (variable renaming)**.** *Let* E *be an expression. A substitution* θ *is called a* variable renaming *iff all the following applies:*

- $\forall X \in vars(E)$ *exists a variable* Y *s.t.* $\theta(X) = Y$*,*
- $vars(E) \cap vars(E\theta) = \emptyset$*,*
- $\forall X \in vars(E)\ \forall Y \in vars(E)\ .\ X \neq Y \Rightarrow \nexists Z \in vars(E\theta)$ *s.t.* $X \mapsto Z \in E\theta \wedge Y \mapsto Z \in E\theta$*.*

Definition 2.2.5 (answer)**.** *Let* P *be a definite program and let* $G = \leftarrow A_1, \ldots, A_n$ *be a definite goal. An* answer *for* $P \cup \{G\}$ *is a substitution* θ *iff* $P \models \forall((A_1 \wedge \ldots \wedge A_n)\theta)$*.*

For example, given a program $P = \{p(a) \leftarrow\}$ and a goal $G = \leftarrow p(X)$, the substitution $\{X \mapsto a\}$ is an answer, but $\{X \mapsto b\}$ is not.

Answers can be calculated based on the concepts of resolution and unification. These concepts are defined below, together with some preliminary definitions.

Definition 2.2.6 (instance)**.** *A term* t *is* more general *than another term* s *(or* s *is an* instance *of* t*), denoted by* $t \leq s$*, if* $\exists\theta.\ t\theta = s$*.*

For example, let $F = p(a, X, Y)$ and $\theta = \{X \mapsto b, Y \mapsto c\}$, then $F\theta = p(a, b, c)$, i.e., $p(a, b, c)$ is an instance of $p(a, X, Y)$. Note that there may exist many instances of a given term, for instance $p(a, b, Y)$ is also an instance of $p(a, X, Y)$.

Definition 2.2.7 (variant)**.** *Two terms* t *and* t' *are* variants*, denoted* $t \approx t'$*, if both* $t \leq t'$ *and* $t' \leq t$*.*

If t *and* t' *are variants then there exists a variable renaming* ρ *such that* $t\rho = t'$*.*

For example, $p(a, X)$ and $p(a, Y)$ are variants since $p(a, X) \leq p(a, Y)$ and $p(a, Y) \leq p(a, X)$.

Definition 2.2.8 (unifier, generalization)**.** *Let S be a finite set of simple expressions. A substitution θ is called a* unifier *for S if $S\theta$ is a singleton, i.e., a set containing a unique element.*

A unifier θ is called most general unifier *(mgu) for S, if for each unifier σ of S, there exists a substitution γ such that $\sigma = \theta\gamma$.*

A generalization *of a set of terms $\{t_1, \ldots, t_n\}$ is another term t such that $\exists\theta_1, \ldots, \theta_n$ with $t_i = t\theta_i$, $i = 1, \ldots, n$.*

A generalization t is the most specific generalization *(msg) of $\{t_1, \ldots, t_n\}$ if for every other term t' s.t. t' is a generalization of $\{t_1, \ldots, t_n\}$, $t' \leq t$.*

For example, $\{p(a, f(X)), p(Y, b)\}$ is not unifiable because the second arguments cannot be unified. However, $\{p(a, f(X)), p(Y, Z)\}$ is unifiable since $\theta' = \{Y \mapsto a, X \mapsto b, Z \mapsto f(b)\}$ is a unifier. A most general unifier is $\theta = \{Y \mapsto a, Z \mapsto f(X)\}$. Note that $\theta' = \theta\{X \mapsto b\}$

From the definition of an *mgu* follows that if θ and σ are both *mgu's* of a set of expressions $\{E_1, \ldots, E_n\}$, then $E_i\sigma$ is a variant of $E_i\theta$. In [86] is shown that each $E_i\sigma$ can be obtained from $E_i\theta$ by simply renaming variables.

Unifiability of a set of expressions is decidable, and there exist efficient algorithms for calculating the *mgu* of two given terms [7, 86].

Given a set of clauses $\{Cl_1 = H_1 \leftarrow B_1, \ldots, Cl_n = H_n \leftarrow B_n\}$, $n \geq 0$, we denote by $instantiate(\{Cl_1, \ldots, Cl_n\}, A)$ the set of clauses $\{Cl_1\theta_1, \ldots, Cl_n\theta_n\}$ where each $\theta_i = mgu(H_i, A)$.

We can now define *SLD-resolution*, which is based on the resolution principle [112], and is a special case of *SL-resolution* [64]. Each SLD-derivation employs a *computation rule* to select an atom within a goal for its evaluation.

Definition 2.2.9 (computation rule)**.** *A* computation rule *is a function $\mathcal{R}$ from goals to atoms. Let G be a goal of the form $\leftarrow A_1, \ldots, A_R, \ldots, A_k$, $k \geq 1$. If $\mathcal{R}(G) = A_R$ we say that A_R is the* selected *atom in G.*

Definition 2.2.10 (derivation step)**.**

Let G be $\leftarrow A_1, \ldots, A_R, \ldots, A_k$. Let $\mathcal{R}$ be a computation rule and let $\mathcal{R}(G) = A_R$. Let $C = H \leftarrow B_1, \ldots, B_m$ be a renamed apart clause in P. Then G'

is derived *from G and C via $\mathcal{R}$ if the following conditions hold:*

$$\theta = mgu(A_R, H)$$

$$G' \text{ is the goal } \leftarrow (A_1, \ldots, A_{R-1}, B_1, \ldots, B_m, A_{R+1}, \ldots, A_k)\theta$$

G' is also called a resolvent *of G and C.*

Definition 2.2.11 (complete SLD-derivation)**.** *Given a program P and a goal G, a* complete SLD derivation *for $P \cup \{G\}$ consists of a possibly infinite sequence $G = G_0 : G_1 : G_2 : \ldots$ of goals, a sequence $C_1 : C_2 : \ldots$ of properly renamed apart clauses of P, and a sequence $\theta_1 : \theta_2 : \ldots$ of* mgus *such that each G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1}.*

Definition 2.2.12 (SLD-refutation)**.** *An* SLD-refutation *of $P \leftarrow \{G\}$ is a finite complete SLD-derivation of $P \leftarrow \{G\}$ which has an empty clause as the last goal of the derivation.*

A derivation step can be non-deterministic when A_R unifies with several clauses in P, giving rise to several possible SLD derivations for a given goal.

SLD derivations can be organized in *SLD trees*.

Definition 2.2.13 (SLD-tree)**.** *A complete* SLD-tree *for $P \cup \{G\}$ is a labelled tree which satisfies:*

- *Each node of the tree is labelled with a definite goal along with an indication of the selected atom.*
- *The root node is labelled with G.*
- *Let $\leftarrow A_1, \ldots, A_R, \ldots, A_k$ be the label of a node in tree, and let A_R be the atom selected by the computation rule $\mathcal{R}$. Then for each clause $A \leftarrow B_1, \ldots, B_n$ in P such that A_R and A are unifiable the node has a child labelled with*

  $\leftarrow (A_1, \ldots, A_{R-1}, B_1, \ldots, B_n, A_{R+1}, \ldots, A_k)\theta$

  where θ is the mgu *of A_R and A.*
- *Nodes labelled with the empty goal have no children. We graphically represent an empty goal with the symbol $\square$.*

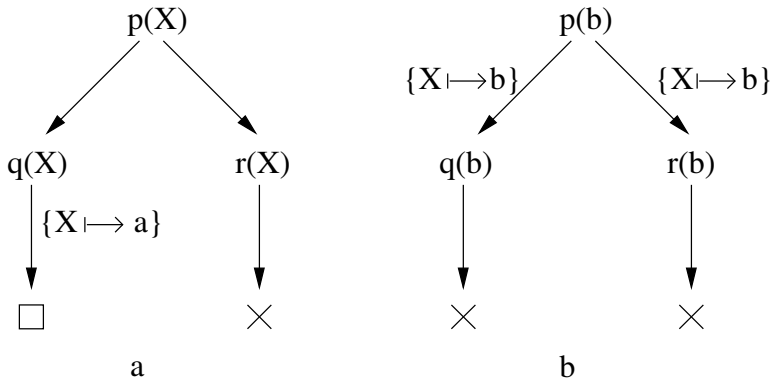

Figure 2.1: SLD-trees for Example 2.2.14

Every branch of a complete SLD-tree corresponds to a complete SLD-derivation. In graphical representations of SLD-trees, selected atoms are underlined.

A finite derivation $G = G_0, G_1, G_2, \ldots, G_n$ is called *successful* if G_n is an empty clause. In that case $\theta = \theta_1\theta_2 \ldots \theta_n$ is called the *computed answer* for goal G. Such a derivation is called *failed* if it is not possible to perform a derivation step with G_n.

Example 2.2.14. *Let us take the program* $P = \{p(X) \leftarrow q(X), p(X) \leftarrow r(X), q(a) \leftarrow\}$. *Given a goal* $G = \{\leftarrow p(X)\}$, *then a possible SLD-tree for* $P \cup \{G\}$, *with computed answer* $\{X \mapsto a\}$, *is shown in Figure 2.1(a). In this figure, successful derivations are represented with* $\square$, *while failed derivations are represented with* $\times$.

Given a goal $G' = \{\leftarrow p(b)\}$, *then a possible SLD-tree for* $P \cup \{G'\}$ *is shown in Figure 2.1(b). There are no successful derivations in this case.*

Chapter 3

Partial Evaluation

In this chapter we provide the basics of *partial evaluation* of logic programs. For a more detailed discussion we refer the reader to works such as [85, 44].

3.1 Basics of Partial Evaluation

The main aim of partial evaluation is to specialize a program w.r.t. part of its input, which is known as the *static data*, the idea being that once the rest of the input (*dynamic data*) is provided, the specialized program—also known as *residual program*—will be more efficient than the original one, since those computation steps which only depend on the static data are performed at specialization time. Thus, in order to obtain the residual program P' of an input program P, a partial evaluator *executes* those parts of P which depend only on the static input S, and *generates residual code* for those parts of P which require the dynamic input D. This process is also called *mixed computation* [38].

Partial evaluation has been applied in a good number of programming paradigms such as imperative programming [6, 5], functional programming [60, 59], logic programming [44, 101], functional logic programming [4, 1], and term rewriting systems [12]. In this thesis we concentrate on partial evaluation of logic programs.

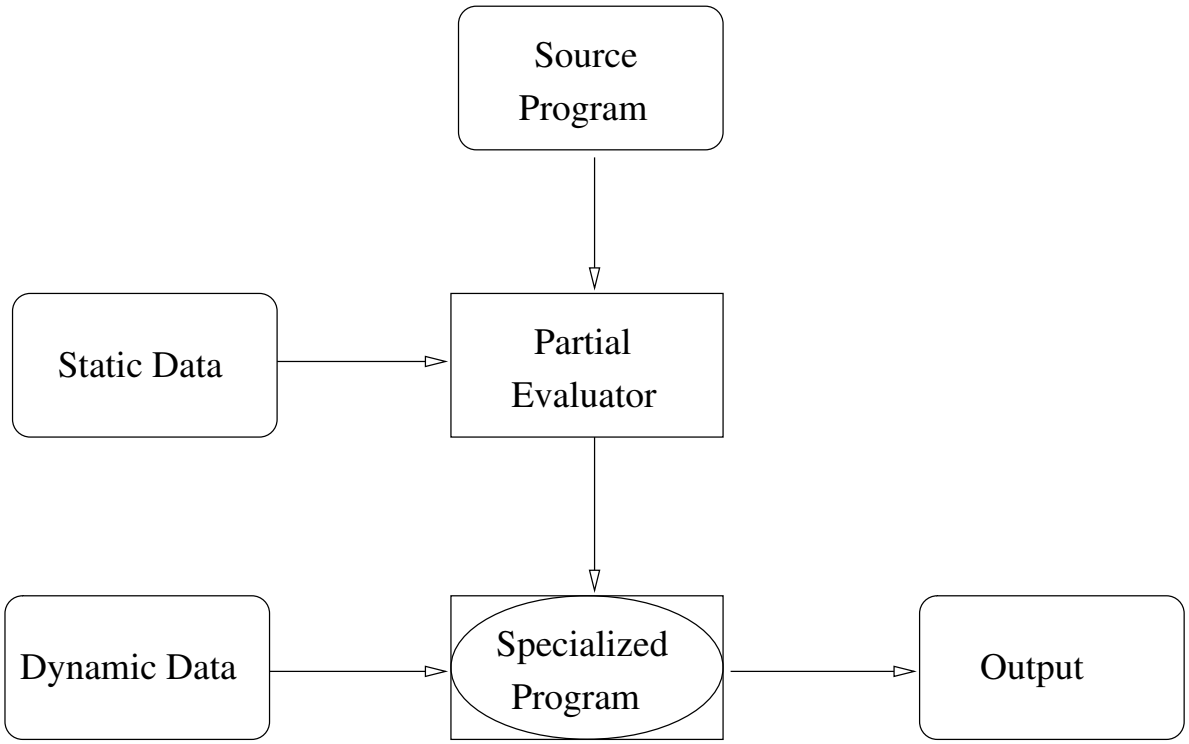

Figure 3.1: An Online Partial Evaluator

3.1.1 Offline vs Online Partial Evaluation

Partial evaluation can be performed in an *online* or *offline* manner. In *online* partial evaluation, the static data is used in order to compute parts of the specialized program as *early* as possible, taking decisions "on the fly" [61, 10, 114, 126]. This process is illustrated in Figure 3.1.

In *offline* partial evaluation [61, 13, 25], the specialization process is split into two phases:

- First, a *binding-time analysis* (*BTA*) is performed which, given a program and an approximation of the input available for specialization, approximates all values within the program during specialization time, and generates an *annotated program*.

- Then, a (simplified) *specialization phase* takes place, which is guided by the annotations generated by the BTA.

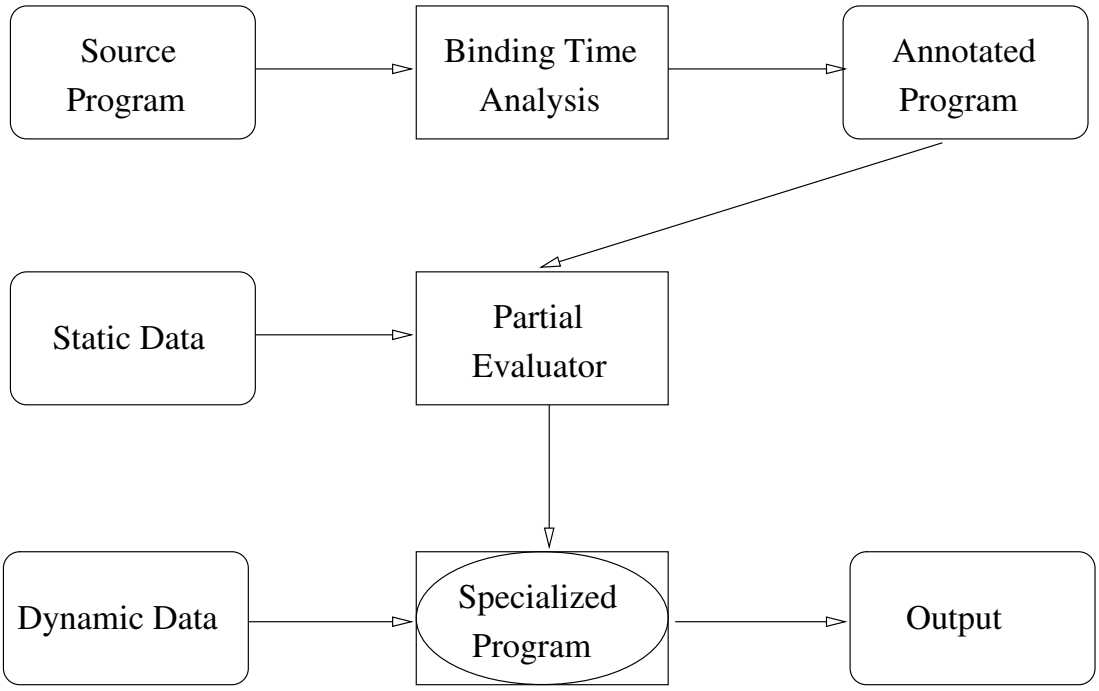

Figure 3.2: An Offline Partial Evaluator

This approach is illustrated in Figure 3.2 and is called *offline* because most control decisions are taken beforehand.

One of the main advantages of the offline approach is the *efficiency of the specialization process*. Once the annotations have been derived, the specializer is relatively simple and can be made to be very efficient. These annotations can be user-provided, and sometimes almost fully automatic. The online approach, on the other hand, is *fully automatic*. Also, the offline approach can only use *more restricted* specialization strategies.

3.2 Partial Evaluation of Logic Programs

When performing partial evaluation of logic programs[1], the static input is a *partially instantiated* goal G. In logic programming, one can still execute a program

[1]The term *partial deduction* [62] is usually used when referring to partial evaluation of *pure* logic programs, i.e., programs without extra-logical features such as cuts, bindings, etc [35]. However, we stick to the term partial evaluation since the framework presented in this work considers these extra-logical features, and is thus oriented to (full) Prolog programs.

P for G and (try to) construct a SLD-tree for $P \cup \{G\}$. However, since G is partially instantiated, this tree is usually infinite, so ordinary evaluation will often not terminate, and we need a more refined approach to partially evaluate logic programs.

Partial evaluation of logic programs is traditionally presented in terms of SLD semantics, which has been introduced in Chapter 2. In partial evaluation, SLD semantics is extended in order to also allow *incomplete derivations*, which are finite derivations of the form $G = G_0, G_1, G_2, \ldots, G_n$ and where no atom is selected in G_n for further resolution. This is needed in order to avoid (local) non-termination of the specialization process. Thus, a SLD-derivation can be successful, failed, incomplete or infinite. The substitution $\theta = \theta_1\theta_2 \ldots \theta_n$ is called the *computed answer substitution* for goal G.

An *incomplete SLD-tree* is defined in the same way as a complete SLD-tree, but possibly containing incomplete derivations. This means that in addition to success and failure leaves, it can also contain *dangling* leaves which correspond to goals which have not been further unfolded, i.e., leaves where no literal has been selected for further derivation. A SLD-tree is called *trivial* iff its root is a dangling leaf.

In short, in order to compute a *partial evaluation* (PE) [85], given an input program and a set of atoms (goal), the first step consists in computing finite incomplete SLD trees for these atoms. Then, a set of *resultants* or residual rules are systematically extracted from the SLD trees.

Definition 3.2.1 (resultant)**.** *Let P be a program, let $\leftarrow G$ be a goal, and let D be a finite SLD-derivation of $P \cup \{\leftarrow G\}$ with computed answer θ and resolvent $\leftarrow B$. Then the formula $G\theta \leftarrow B$ is a* resultant *of D.*

Definition 3.2.2 (partial evaluation)**.** *Let P be a definite program and let A be an atom. Let τ be a SLD tree for $P \cup \{\leftarrow A\}$, and let $\leftarrow G_1, \ldots, \leftarrow G_n$ be goals chosen from the non-root nodes of τ such that there is exactly one goal from each non-failing branch of τ. Let $\theta_1, \ldots, \theta_n$ be the computed answers of the derivations from $\leftarrow A$ to $\leftarrow G_1, \ldots, G_n$ respectively. Then the set of resultants $\{A\theta_1 \leftarrow G_1, \ldots, A\theta_n \leftarrow G_n\}$ is called a partial evaluation of A in P.*

If $\mathcal{A}$ is a finite set of atoms, then a partial evaluation of $\mathcal{A}$ in P is the union of the partial evaluations of the elements of $\mathcal{A}$.

Given a program P and an atom A, there exist in general infinitely many different partial evaluations of A in P [44]. An *unfolding rule* is a fixed rule for generating resultants.

Definition 3.2.3 (unfolding rule)**.**

Given a program P and an atom A, an unfolding rule *computes a SLD tree for $P \cup \{\leftarrow A\}$.*

We use $U(P, A) = \tau$ to denote the fact that the unfolding rule U, when applied to an atom A in program P, returns the SLD tree τ.

We now introduce the concepts of *closedness* and *independence*, necessary to establish correctness of partial evaluation.

Definition 3.2.4 (closed)**.** *Let S be a set of first-order formulas and let A be a finite set of atoms. Then S is A-*closed *if every atom in S containing a predicate symbol occurring in A is an instance of an atom in A.*

Definition 3.2.5 (independence)**.** *Let A be a set of atoms. Then A is independent if no two atoms in A have a common instance.*

The central result proved in [85] is the following theorem about correctness of partial evaluation.

Theorem 3.2.6. *Let P be a definite program, and let $\mathcal{A}$ be an independent set of atoms. Let P' be a partial evaluation of $\mathcal{A}$ in P. Then for all goals G such that $P' \cup \{G\}$ is $\mathcal{A}$-*closed

- *$P \cup \{G\}$ has a SLD-refutation with computed answer θ iff $P' \cup \{G\}$ has a SLD-refutation with computed answer θ.*
- *$P \cup \{G\}$ has a finitely-failed SLD-tree iff $P' \cup \{G\}$ has a finitely-failed SLD-tree.*

3.2.1 A Greedy Partial Evaluation Algorithm

Algorithm 1 shows a greedy algorithm for performing partial evaluation of a given program. In this algorithm, besides an *unfolding rule U*, an *abstract operation G* is used, and whose main purposes are:

- to ensure termination of the algorithm, and
- to satisfy the independence requirement described above.

Algorithm 1 Partial Evaluation Algorithm (PE)

Input: Program P
Input: Set of atoms of interest $\mathcal{A}$
Input: An abstraction rule G
Input: An unfolding rule U
Output: A partial evaluation for P and $\mathcal{A}$, encoded by H_i

1: $i = 0$
2: $H_0 = \emptyset$
3: $\mathcal{A}_0 = \mathcal{A}$
4: **repeat**
5: $\quad A_i = Select(\mathcal{A}_i)$
6: $\quad A'_i = G(H_i, A_i)$
7: $\quad \tau_i = U(P, A'_i)$
8: $\quad H_{i+1} = H_i \cup \{\langle A_i, A'_i \rangle\}$
9: $\quad \mathcal{A}_{i+1} = (\mathcal{A}_i - \{A_i\}) \cup \{A \in leaves(\tau_i) \mid \forall\ \langle B, \_ \rangle \in H_{i+1}\ .\ B \not\approx A\}$
10: $\quad i = i + i$
11: **until** $\mathcal{A}_i = \emptyset$

The potential queries to the program are represented by the set of atoms $\mathcal{A}$. In each iteration of the algorithm, an atom A_i from $\mathcal{A}_i$ is selected (line 5). Then, global control and local control as defined by the G and U rules, respectively, are applied (lines 6 and 7). This builds an SLD-tree for A'_i, a generalization of A_i as determined by G, using the predefined unfolding rule U. Once the SLD-tree τ_i is computed, the leaves in its resultants, i.e., the atoms in the residual code for A'_i are collected by the function *leaves*. Those atoms in $leaves(\tau_i)$ which are not a variant of an atom handled in previous iterations of the algorithm are added to the set of atoms to be considered ($\mathcal{A}_{i+1}$). We use $B \approx A$ to denote that B and A are *variants*, i.e., they are equal modulo variable renaming. The algorithm finishes when $\mathcal{A}_i$ becomes empty.

The specialized program P' corresponds to

$$P' = \bigcup_{\langle A,A' \rangle \in H_i} resultants(U(P, A')).$$

where *resultants* extracts the residual rules from the SLD trees resulting from unfolding each of the abstracted atoms in all tuples of H_i.

Note that this algorithm differs from those in [44, 71] in that once an atom A_i is abstracted into A'_i, code for A'_i will be generated, and it will not be abstracted any further no matter which other atoms are handled in later iterations of the algorithm. As a result, the set of atoms for which code is generated are not guaranteed to be *independent*. However, the pairs in H_i uniquely determine the version used at each program point. Since code generation produces a new predicate name per entry in H_i, independence is guaranteed, and thus the specialized program will not produce more solutions than the original one.

The ECCE system [84] can be made to behave as Algorithm 1 by setting the *parent abstraction* flag to *off*.

3.3 Control and Termination of Partial Evaluation

As mentioned before, in partial evaluation we can distinguish two levels of control [44], the so-called *global control*, in which one decides which atoms are to be partially evaluated, and the *local control*, in which one constructs the (possibly incomplete) SLD-tree for each atom in the set of atoms being handled by the partial evaluation algorithm. Thus, we must consider two distinct questions about termination:

- Termination of the iterative algorithm, also known as global termination.
- Termination of the unfolding rule U, better known as local termination.

We briefly discuss these issues below.

3.3.1 Local Termination

In order to ensure the local termination of the partial evaluation algorithm while producing useful specializations, the unfolding rule must incorporate some non-trivial mechanism to stop the construction of SLD trees. Nowadays, well-founded

orderings (wfo) [16, 88] and well-quasi orderings (wqo) [117, 81] are broadly used in the context of on-line partial evaluation techniques (see, e.g., [44, 84, 117]).

We now formally define well-founded orderings and well-quasi orderings.

Definition 3.3.1 (s-poset)**.** *A* strict partial order *on a set S is an anti-reflexive, anti-symmetric and transitive binary relation on $S \times S$. A partially strictly ordered set, or* s-poset *$S, >_S$, consists of a set S and a strict partial order $>_S$ on S.*

Definition 3.3.2 (wfo)**.** *An s-poset $S, >_S$ is* well-founded *iff there is no infinite sequence of elements $s_1 : s_2 : \ldots$ in S such that $s_i > s_{i+1}$ for all $i \geq 1$. The order $>_S$ is called a* well-founded order (wfo) *on S.*

Definition 3.3.3 (poset)**.** *A* (non-strict) partial order *on a set S is a reflexive and transitive binary relation on $S \times S$. A partially ordered set, or* poset *$S, \geq_S$, consists of a set S and a partial order $\geq_S$ on S.*

Definition 3.3.4 (wqo)**.** *A poset $S, \leq_S$ is* well-quasi-ordered (wqo) *iff for any infinite sequence of elements $s_1 : s_2 : \ldots$ in S there are $i < j$ such that $s_i \leq_S s_j$. The order $\leq_S$ is called a* well-quasi order (wqo) *on S.*

It is well known that the use of wfos and wqos allows the definition of *admissible* sequences which are always finite. Intuitively, derivations are expanded as long as there is some evidence that interesting computations are performed and also guaranteed to terminate (according to the selected ordering).

Intuitively, a sequence of elements $s_1 : s_2 : \ldots$ in S is called *admissible with respect to an order $\leq_S$* [16] iff there are no $i < j$ such that $s_i \leq_S s_j$. If the order is a wqo, given a derivation $G_1, G_2, \cdots, G_{n+1}$ in order to decide whether to evaluate G_{n+1} or not, we check that the selected atom in G_{n+1} is strictly smaller than any previous (comparable) selected atom. A more formal definition is provided below.

Definition 3.3.5 (admissible)**.** *Let $\leq_S$ be a wqo. We denote by $Admissible(A, (A_1, \ldots, A_n), \leq_S)$, with $n \geq 0$, the truth value of the expression $\forall A_i,\ i \in \{1, \ldots, n\}\ : A_i \not\leq_S A$. In a wfo, it is sufficient to verify that the selected atom is strictly smaller than the previous comparable one (if one exists). Let $<$ be a wfo, by $Admissible(A, (A_1, \ldots, A_n), <)$, with $n \geq 0$, we denote the truth value of the expression $A_n \not< A$ if $n \geq 1$ and true if $n = 0$.*

We will denote by *structural order* a wfo or a wqo (written as $\triangleleft$ to represent any of them). Among the structural orders, well-quasi orderings have proved to be very powerful in practice. In particular, the *homeomorphic embedding* [67] ordering is the wqo we will use in our experiments. Informally, an atom t_1 *embeds* atom t_2 if t_2 can be obtained from t_1 by deleting some operators, e.g., $\mathtt{s}(\underline{\mathtt{s}}(\underline{\mathtt{U}}+\mathtt{W})\underline{\times}(\underline{\mathtt{U}}\underline{+}\mathtt{s}(\underline{\mathtt{V}})))$ embeds $\mathtt{s}(\mathtt{U}\times(\mathtt{U}+\mathtt{V}))$. The interested reader is referred to [37, 81] where a detailed description of homeomorphic embedding can be found.

3.3.2 Global Termination

In addition to local control, an *abstraction operator* is applied to properly add the atoms in the right-hand sides of resultants to the set of atoms to be partially evaluated (line 6 of Algorithm 1). This abstraction operator performs the *global control* and is in charge of guaranteeing that the number of atoms which are generated remains finite by replacing atoms by more general ones, i.e., by losing precision in order to guarantee termination.

Definition 3.3.6 (abstraction)**.** *Let $\mathcal{A}$ be a set of atoms. The set of atoms $\mathcal{A}'$ is an* abstraction *of $\mathcal{A}$ iff every atom of $\mathcal{A}$ is an instance of an atom in $\mathcal{A}'$.*

The use of an abstraction operator does not guarantee global termination in itself. Some abstraction operators guarantee termination by losing some precision. For instance, we could impose a finite number of atoms in $\mathcal{A}$ and apply the *most specific generalization (msg)* operator to enforce not exceeding such limit.

Throughout this thesis, we will use in most of examples and experiments the following global control rules:

id: is the identity abstraction rule, i.e., for any atom A, $G(A) = A$. This rule is equivalent to not performing any abstraction at all, and thus it does not guarantee termination of the partial evaluation algorithm. It is therefore only used in cases where global termination is achieved, even when no abstraction is performed.

dynamic: is the most abstract global control rule possible, which abstracts away the value of all arguments of the atom and replaces them with distinct variables. For example, $G(p(1, X, [a, b]) = p(X', Y', Z')$

hom_emb: is based on homeomorphic embedding [71, 81] and optionally also on global trees [74]. It flags atoms as potentially dangerous (and are thus generalized) when they homeomorphically embed any of the previously visited atoms.

3.4 Unfolding Strategies

In the basic partial evaluation algorithm we assume the existence of an unfolding rule U, which takes an atomic goal and a program, and returns a finite, possibly incomplete SLD tree for them. There are many possible unfolding strategies. Some of them perform better in some situations and worse in others when compared with one another.

In `CiaoPP` [54, 106, 55], an unfolding rule is composed of an *unfolding strategy*, a *computation rule* and a *unfolding branching factor*. We present in this section some of the several unfolding strategies and computation rules available in `CiaoPP`. The ones presented here will be used later for our experiments.

3.4.1 Determinate Unfolding

Determinate unfolding is a quite simple unfolding strategy where unfolding is performed *only* if the current selected atom of a goal matches at most a single clause head of the program. As shown in [44], unfolding determinate goals does not introduce extra computation into a program, even if the selected atom is non-leftmost. However, in many cases this unfolding strategy is too conservative. Also, it is not always guaranteed to obtain finite SLD trees.

Definition 3.4.1 (determinate unfolding)**.** *A SLD tree is (*purely*)* determinate *if each node of the tree has at most 1 child. An unfolding strategy U is* purely determinate *if it returns a determinate SLD-tree for any program P and any goal G.*

A "look-ahead" of a finite number of computation steps can be used to detect further cases of determinacy. For instance, given the program

```
p(X) :- X>0, q(X).
p(X) :- X<=0, r(X).
```

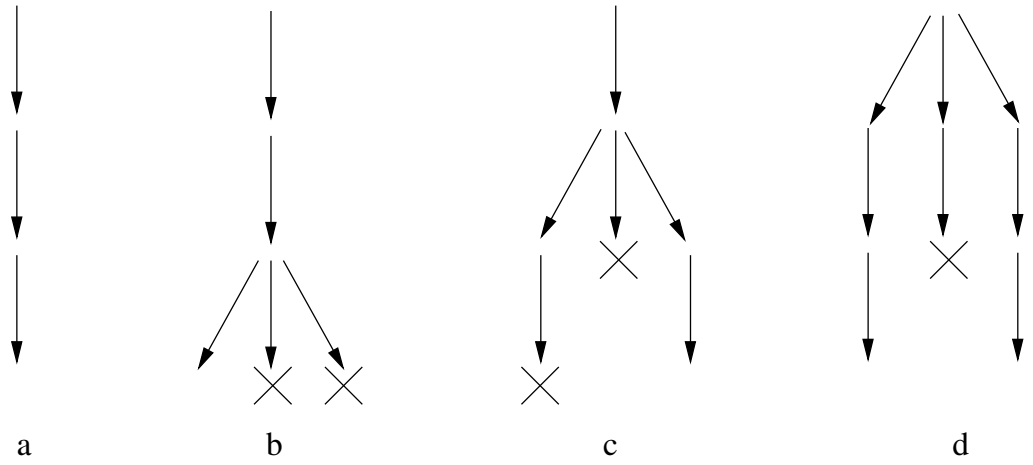

Figure 3.3: Different Unfolding Trees

the goal $\leftarrow$ `p(1)` is determinate after look-ahead of 1 step.

In Figure 3.3 we can see several unfolding trees where failing derivations are marked with $\times$. Tree `a` is purely determinate. Tree `b` is determinate after look-ahead of 1 step. Tree `c` is determinate after look-ahead of 2 steps. Finally, tree `d` is not determinate after look-ahead of 2 steps (although it could become determinate later, but this is an undecidable problem).

3.4.2 One-Step Unfolding

The rule *one_step* is the simplest possible unfolding strategy which always performs just one unfolding step for any atom.

3.4.3 Unfolding Based on Homeomorphic Embedding

It is well-known that imposing some well-founded order on selected atoms guarantees termination while leading to overly eager unfolding [16, 88]. Instead of well-founded orders, it is possible to use well-quasi orders as well [11].

Intuitively, and in order to ensure local termination, if an atom we want to select is homeomorphically embedded by one of its ancestor then we have to select a different atom or we have to stop unfolding. State-of-the-art unfolding strategies allow performing ordering comparisons over *subsequences* of the full sequence of the selected atoms of a derivation by organizing atoms in a *proof tree*

[15]. This allows considering the embedding relation with the *covering ancestors* of the selected atom, achieving further specialization in many cases while still guaranteeing termination.

`CiaoPP` provides several unfolding strategies based on homeomorphic embedding. Throughout this work we will extensively use an unfolding rule called *df_hom_emb_as*. It can handle external predicates safely and can perform non-leftmost unfolding (see below) as long as unfolding is safe (see [2]) and *local* (see [106]).

3.4.4 Computation Rules

Besides the unfolding strategy chosen for partially evaluating a program, we need to define the computation rule that is going to be used. Although `CiaoPP` offers several posibilities, there are two important classes of computation rules that we will use throughout this thesis:

leftmost This is the trivial computation rule which always returns the *leftmost* atom in a goal. This computation rule is interesting in that it avoids several correctness and efficiency issues in the context of PE of full Prolog programs [2, 44].

non-leftmost Informally, given a program P and a goal $\leftarrow A_1, \ldots, A_n$, it can happen that the leftmost atom A_1 cannot be selected for unfolding due to several circumstances. Among others, if A_1 is an atom for a predicate defined in P (thus the code is available to the partial evaluator) it can happen that:

- unfolding A_1 endangers termination (for example, A_1 may homeomorphically embed some selected atom in its sequence of covering ancestors), or
- the atom A_1 unifies with several clause heads (deterministic unfolding strategies do not unfold non-deterministically for atoms other than the initial query).

If A_1 is an atom for an external predicate whose code is not present nor available to the partial evaluator, it can happen that

- A_1 is not sufficiently instantiated so as to be executed at this moment.

  In cases like this it is interesting to select a *non-leftmost* atom. It is well-known that the ability of performing *non-leftmost* unfolding is essential in partial evaluation in some cases for the satisfactory propagation of static information (see, e.g., [71]).

For logic programs without impure predicates, non-leftmost unfolding is sound thanks to the independence of the computation rule (see for example [86])[2]. Unfortunately, non-leftmost unfolding poses several problems in the context of *full* Prolog programs with *impure* predicates, where such independence does not hold anymore. For instance, `ground/1` is an *impure* predicate since, under LD resolution[3], the goal `ground(X),X=a` fails whereas `X=a,ground(X)` succeeds with computed answer X/a. Those executions are not equivalent and, thus, the independence of the computation rule does no longer hold. As a result, given the goal $\leftarrow$ `ground(X),X=a`, if we allow the non-leftmost unfolding step which binds the variable `X` in the call to `ground(X)`, the goal will succeed at specialization time, whereas the initial goal fails in LD resolution at run-time. The above problem was early detected [115] and it is known as the problem of *backpropagation of bindings*. Also, *backpropagation of failure* is problematic in the presence of impure predicates. For instance, $\leftarrow$ `write(hello),fail` behaves differently from $\leftarrow$ `fail`. For a thorough discussion of these problems, see for example [2].

`CiaoPP` provides a parameter that is used to decide when unfolding should be avoided, when using a non-leftmost computation rule. This parameter is called *unf_bra_fac*, standing for *unfolding branching factor*, and its value is a natural number. For example, if set to i, $i > 0$, it means that if the selected atom unifies with j clause heads, $j \leq i$, then unfolding continues, otherwise it stops. We reserve the value 0 for indicating that *no limit* is imposed on the branching factor. If *unf_bra_fac* is different from 1, it is not guaranteed that the residual program will execute fewer resolution steps.

In some chapters of this thesis, we will use *hom_emb_aggr* to refer to the unfolding rule using an unfolding strategy based on homeomorphic embedding for

[2]However, non-deterministic unfolding of nonleftmost atoms can degrade efficiency.

[3]LD resolution is a case of SLD resolution where the selection rule is set to the left-to-right selection rule of Prolog [35, 118].

flagging possible non-termination, and using 0 as an unfolding branching factor, and *hom_emb_cons* when using 1 as the unfolding branching factor. In both cases, the computation rule will allow selecting non-leftmost atoms. Note that when *one_step* is being used as an unfolding strategy, then neither the computation rule nor the unfolding branching factor make any difference.

In the literature, and also throughout this thesis, we will use *(non-)leftmost unfolding* to denote that we are performing unfolding using a *(non-)leftmost computation rule*.

3.5 Partial Evaluation of *Full* Prolog Programs

Most of real-life Prolog programs use predicates which are not defined in the program (module) being developed. We will refer to such predicates as *external*. Examples of external predicates are the traditional "builtin" predicates such as arithmetic operations (e.g., `is/2`, `<`, `=<`, etc.) or basic input/output facilities. We will also consider as external predicates those defined in a different module, predicates written in another language, etc.

Although some builtins, usually taken to be higher-order, such as `map/3`, can be mapped to pure definite (first-order) logic programs [125], most builtins like `assert/1` and `retract/1` are extra-logical and ruin the declarative nature of the underlying program. In this section we explain the difficulties that such *external* predicates pose during partial evaluation.

3.5.1 Performing Derivation Steps over External Predicates

When an atom A, such that $pred(A) = p/n$ is an external predicate, is selected during partial evaluation, it imposes several difficulties for performing a derivation step.

- First, we may not have the code defining p/n and, even if we have it, the derivation step may introduce in the residual program calls to predicates which are private to the module M where p/n is defined.

- In spite of this, if the executable code for the external predicate p/n is available, and under certain conditions, it can be possible to fully evaluate calls to external predicates at specialization time. We use $\mathsf{Exec}(Sys, M, A)$ to denote the execution of atom A on a logic programming system Sys, in which the module M where the external predicate p/n is defined has been loaded. In the case of logic programs, $\mathsf{Exec}(Sys, M, A)$ can return zero, one, or several computed answers for $M \cup A$ and then execution can either terminate or loop.

We will use substitution sequences [21] to represent the outcome of the execution of external predicates. A *substitution sequence* is either

- a finite sequence of the form $\theta_1 : \ldots : \theta_n$, $n \geq 0$, or
- an incomplete sequence of the form $\theta_1 : \ldots : \theta_n : \bot$, $n \geq 0$, where $\bot$ indicates that the execution loops, or
- an infinite sequence $\theta_1 : \ldots : \theta_i : \ldots$, $i \in \mathbb{N}^*$, where $\mathbb{N}^*$ is the set of positive natural numbers.

We say that an execution *universally terminates* if $\mathsf{Exec}(Sys, M, A) = \theta_1 : \ldots : \theta_n$, $n \geq 0$.

In addition to producing substitution sequences, it can be the case that the execution of atoms for (external) predicates produces other outcomes such as side-effects, errors, and exceptions. Note that this precludes in principle the evaluation of such atoms to be performed at partial evaluation time, since those effects need to be performed at run-time.

The notion of *evaluable* atom was introduced in [106], in order to capture the requirements which allow executing external predicates at partial evaluation time.

Definition 3.5.1 (evaluable)**.** *Let A be an atom such that $pred(A) = p/n$ is an external predicate defined in module M. We say that A is* evaluable *in a logic programming system Sys if $\mathsf{Exec}(Sys, M, A)$ satisfies the following conditions:*

1. *it universally terminates*
2. *it does not produce side-effects*

3. *it does not issue errors*

4. *it does not generate exceptions*

We also say that an expression E is evaluable if

1. *E is an evaluable atom, or*

2. *E is a conjunction of evaluable expressions, or*

3. *E is a disjunction of evaluable expressions.*

3.6 Partial Evaluation: an Example

As an example, let us take the `Ciao` program in Listing 3.1, where `exp(A,B,C)` returns in `C` the result of computing A^B.

Listing 3.1: The `exp/3` Example

```
:- module(_,[exp/3],[assertions]).
:- pred exp(+Base,+Exp,-Res).

exp(Base,Exp,Res):- exp_ac(Exp,Base,1,Res).

exp_ac(0,_,Res,Res).
exp_ac(Exp,Base,Tmp,Res):-
        Exp > 0,
        Exp1 is Exp - 1,
        NTmp is Tmp * Base,
        exp_ac(Exp1,Base,NTmp,Res).
```

Note that this program is not a pure logic program, since it uses builtins such as `is/2`. A *pure* predicate does not distinguish between input and output arguments, and any of them can be instantiated when calling the predicate. However, the situation is different when we consider impure predicates. In this case, the input to predicate `exp(A,B,C)` is represented by the arguments `A` and `B`, while `C` is the output. This means that when calling `exp/3` at runtime, both `A` and `B` must be instantiated, in this case to numbers. As shown in the program, the

direction of arguments can be expressed in the program by using `CiaoPP` assertions [107]. In this example we use *modes* to indicate with `+` and `-` that the first two arguments are input, while the third argument is output.

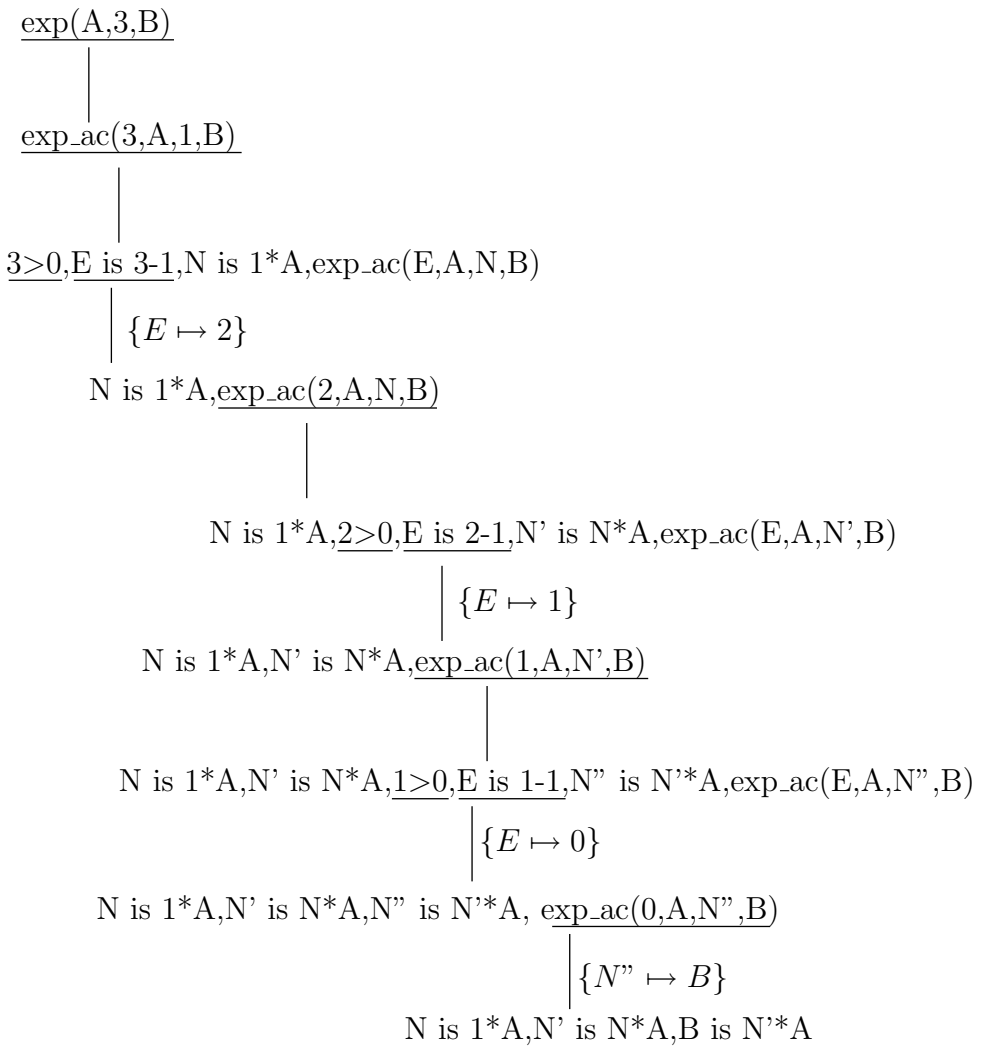

Figure 3.4: Unfolding Tree for `exp(A,B,C)` When `B` Is Known

If we know in advance the value of some input argument of `exp/3`, we could use partial evaluation to specialize this predicate w.r.t. the known input. For example, let us say that we know that the value of `B` (the exponent) is `3`, then we can specialize this program and obtain the (more efficient) residual program shown in Listing 3.2.

Listing 3.2: Residual Code of the `exp/3` Example

```
exp(A,3,B) :-  N is 1*A, N' is N*A, B is N'*A.
```

Note that this specialized program can be obtained using a non-leftmost computation rule, and jumping over the builtin `is/3` in Figure 3.4 (e.g. `N is 1*A`,

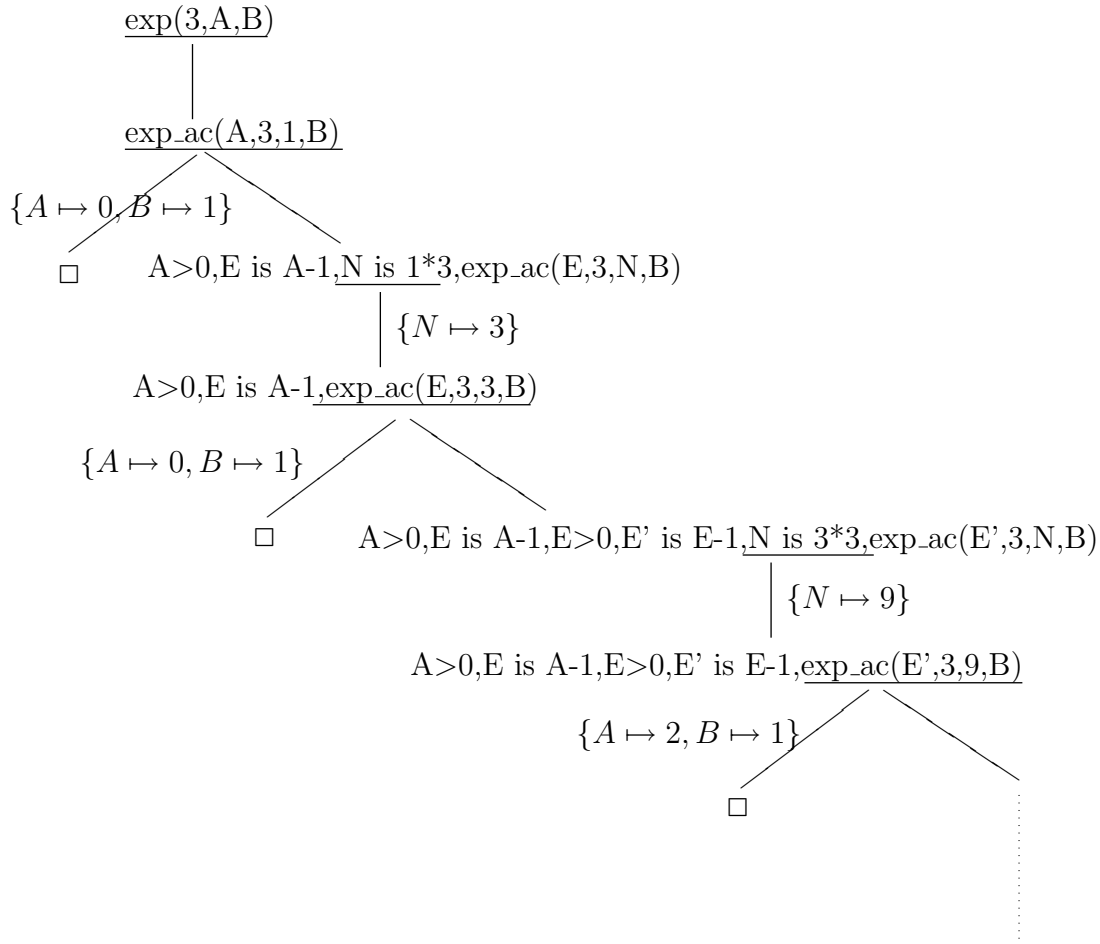

Figure 3.5: Possibly Infinite Unfolding Tree for `exp(A,B,C)`

`N' is N*A`, ...), which is safe since the `pred` assertion in Listing 3.1 indicate us that `A` (the base) will be instantiated at runtime [2]. In the SLD-tree represented in Figure 3.4, selected atoms are underlined, and sometimes we squeeze the graphical representation (in order to make the tree smaller) by representing two consecutive resolution steps in one node, as done for example when selecting `3>0` and `E is 3-1`.

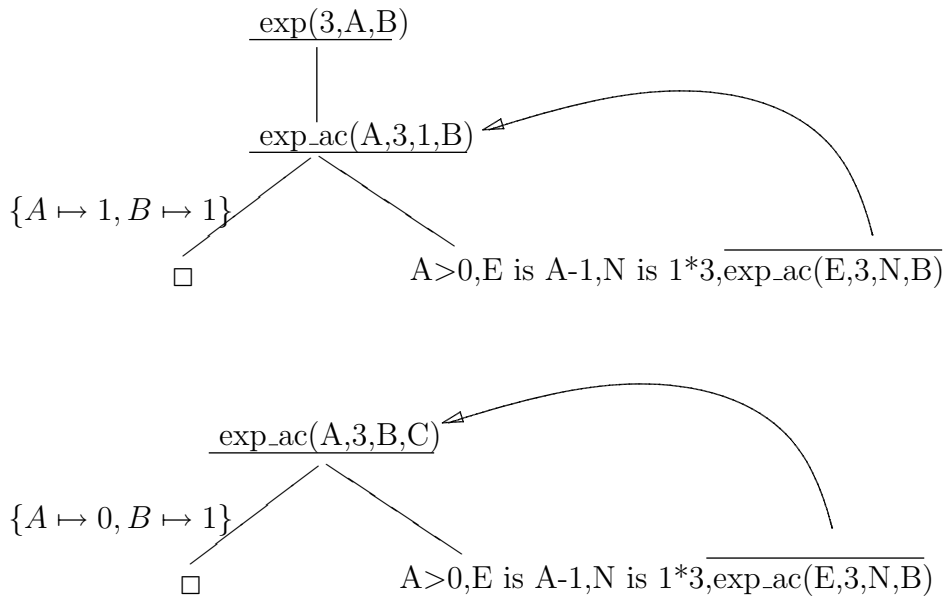

Figure 3.6: Unfolding Tree for `exp(A,B,C)` When `A` Is Known

The specialized program is not only more efficient in terms of speed (the loop of the original program has been wiped out), but also in terms of size of code, and memory taken by the residual program. Obtaining a specialized program that outperforms the original one in several performance aspects represents an ideal situation, but unfortunately not achieved often. In general, we will have a tradeoff between size- and speed-efficiency, as we will see later in this thesis.

However, also note that in some situations partial evaluation cannot be of much help. For instance, let us assume that we know in advance that the value of the base is 3, i.e., `A=3`. If we use a leftmost computation rule then we would stop in `Exp>0` since the value of the exponent is unknown. Using a non-leftmost computation rule we could continue unfolding, as shown in Figure 3.5. However, the unfolding tree is potentially infinite, and the process could continue forever.

If we use an unfolding strategy based on homeomorphic embedding (see Figure 3.6), then we can realize that `exp_ac(E,3,N,B)` embeds its ancestor `exp_ac(A,3,1,B)`(indicated by curved arrows in the figure) and stop unfolding. Later, when unfolding `exp_ac(E,3,N,B)` we would find a similar embedding. From these two SLD-trees, we obtain the residual program shown in Listing 3.3. As can be seen, this code is larger than the original and it does not reduce the

number of computation steps.

Listing 3.3: Residual Code for the `exp/3` Example (II)

```
exp(3,0,1).
exp(3,A,B) :- A>0, C is A-1,
        D is 1*3,  exp_ac_1(C,3,D,B) .

exp_ac_1(0,3,A,A).
exp_ac_1(A,3,B,C) :- A>0, D is A-1,
        E is B*3, exp_ac_1(D,3,E,C) .
```

Part II

Reducing the Size of Specialized Programs

Chapter 4

Removing Superfluous Versions in Polyvariant Specialization

As mentioned in Chapter 3, traditional partial evaluation (PE) of logic programs aims at obtaining code which is as optimized as possible. In general, this is achieved by performing aggressive unfolding at the local control level, and by being as accurate as possible (generalize the least possible) at the global control level, as long as termination is guaranteed [71]. In particular, given a fixed local control rule, different global control rules will have different effects on the polyvariance level of partial evaluation, i.e., the number of versions produced for each procedure. A common heuristic is to allow a high degree of polyvariance as long as termination is not compromised, the idea being that by considering different versions separately, further optimizations may be uncovered. This heuristic makes sense from the point of view of optimizing programs in terms of resolution steps, but it can produce unnecessarily large results, and may even slow down programs due to cache miss effects (see e.g. [121, 28].

The problem of superfluous polyvariance has been studied both in the context of abstract multiple specialization [130, 108] and in the context of partial evaluation of normal logic programs [84]. The common idea is to identify sets of versions which are *equivalent* and replace all occurrences of such versions by a single, canonical, one. This poses two questions:

1. under which conditions can we consider two given versions as equivalent?
2. how can we efficiently check for equivalence?

To address the first question, we need to first accurately define the notion of equivalence of versions. Then, in order to make this comparison in an efficient way, we can use some additional information such as making sure that the versions correspond to the same predicate in the original program, and also use their specialization history, which can be collected through abstractions such as characteristic trees [42, 84] and trace terms [43].

In this chapter, we provide a thorough analysis of these questions, comparing different approaches for controlling polyvariance, and we also extend previous approaches in two ways.

- First, we tackle in an accurate way the case in which programs contain *external predicates*, i.e., predicates whose code is not defined in the program being specialized, and thus it is not available to the specializer. This includes predicates defined in other user modules, library predicates, builtins, predicates implemented in other languages, etc. Note that external predicates differ from regular ones in that they cannot be unfolded using the traditional mechanism, and in that they may have *impure* features.

  We show an extension to traditional *characteristic trees* and *trace terms* which can be used in the presence of calls to external predicates. This extension was first proposed in [94]. Based on this extension, we define sufficient conditions for minimization, which are more accurate than those used in previous work, potentially resulting in a higher degree of minimization.

- Second, previously proposed minimization techniques do not provide any degrees of freedom at the minimization stage. We propose an additional generalization of the notion of equivalence which introduces the possibility of collapsing versions which are not *strictly* equivalent. This is achieved by residualizing certain computations for external predicates which would otherwise be performed at specialization time. This allows automatically trading time for space and is of interest in the context of embedded and pervasive systems, where computing resources and storage are often limited.

A completely different approach to that studied in this chapter is to incorporate within the global control certain heuristics which limit polyvariance based for example on characteristic trees [42, 78, 83]. Such approach has both advantages

```
(1) main(A,B,C,D,E,F,G,H,I,J,K,L,M,N,O):-
      write(C),
      addlists([4,4|A],[0,3|B],[4,7|C]),
      addlists([3,3|D],[1,4|E],[4,7|F]),
      addlists([3,3|G],[1,4|H],I),
      addlists([1,1|J],[3,6|K],L),
      addlists([7,1|M],[1,5|N],O).

(2) addlists([],[],[]).
(3) addlists([A|B],[C|D],[H|T]):-
      H is A+C,
      addlists(B,D,T).
```

Figure 4.1: Adding Pairs of Lists.

and disadvantages. The advantage is that there is no need to perform a post minimization phase, such as that discussed in this chapter. On the other hand, the disadvantage of that approach is that it sometimes produces results which are suboptimal, since the fact that characteristic trees are equal not always means that the corresponding versions should be merged.

We argue that a minimization phase is important in specialization algorithms, since it allows using very accurate global control rules while limiting the risk of generating large residual code with many similar versions. Rather than deciding *a priori* the best global control possible, this technique allows using aggressive control strategies. We can minimize the program *a posteriori* and eliminate those specialized versions which are redundant.

4.1 Polyvariant Specialization: an Example

Example 4.1.1. *In order to see the effects of polyvariance, let us use the example in Figure 4.1. Predicate* `addlists/3` *adds the contents of two lists, using the builtin* `is/2`*. Clauses are numbered for later reference. A possible result of partial evaluation for the initial query* `main/15` *is shown in Figure 4.2.*

```
main(A,B,C,D,E,F,G,H,I,J,K,L,M,N,O) :- write(C),
    addlists_2([4,4|A],[0,3|B],[4,7|C]),
    addlists_3([3,3|D],[1,4|E],[4,7|F]),
    addlists_4([3,3|G],[1,4|H],I),
    addlists_5([1,1|J],[3,6|K],L),
    addlists_6([7,1|M],[1,5|N],O).

addlists_1([],[],[]).
addlists_1([A|B],[C|D],[E|F]) :-
    E is A+C, addlists_1(B,D,F).

addlists_2([4,4],[0,3],[4,7]).
addlists_2([4,4,A|B],[0,3,C|D],[4,7,E|F]) :-
    E is A+C, addlists_1(B,D,F).

addlists_3([3,3],[1,4],[4,7]).
addlists_3([3,3,A|B],[1,4,C|D],[4,7,E|F]) :-
    E is A+C, addlists_1(B,D,F).

addlists_4([3,3],[1,4],[4,7]).
addlists_4([3,3,A|B],[1,4,C|D],[4,7,E|F]) :-
    E is A+C, addlists_1(B,D,F).

addlists_5([1,1],[3,6],[4,7]).
addlists_5([1,1,A|B],[3,6,C|D],[4,7,E|F]) :-
    E is A+C, addlists_1(B,D,F).

addlists_6([7,1],[1,5],[8,6]).
addlists_6([7,1,A|B],[1,5,C|D],[8,6,E|F]) :-
    E is A+C, addlists_1(B,D,F).
```

Figure 4.2: Specialization of `addlists/3` via Partial Evaluation.

Here, we use df_hom_emb_as as the unfolding strategy. As we mentioned in

Chapter 3, this rule is based on homeomorphic embedding [71] and it never performs non-leftmost unfolding steps to the right of a (possibly) impure predicate. This guarantees the correctness of the partial evaluation process even in the presence of impure predicates. Note, however, that the issue of redundant polyvariance may occur for any unfolding strategy. The global control used is hom_emb, which is based on homeomorphic embedding and global trees [74].

Unfolding of `main/15` *only performs one step since the leftmost literal* `write(C)` *has side-effects, and performing non-leftmost unfolding of any other literal may backpropagate bindings (as variables may be aliases) onto* `write(C)`*. Note that one version has been generated for each call to* `addlists/3` *within the body of* `main/15`*, plus one version for the general case. However, the four versions* `addlists_2` *through* `addlists_5` *are indeed equivalent and could be replaced by a single one, resulting in the (smaller) program shown in Figure 4.3.*

```
main(A,B,C,D,E,F,G,H,I,J,K,L,M,N,O) :- write(C),
     addlists_5([4,4|A],[0,3|B],[4,7|C]),
     addlists_5([3,3|D],[1,4|E],[4,7|F]),
     addlists_5([3,3|G],[1,4|H],I),
     addlists_5([1,1|J],[3,6|K],L),
     addlists_6([7,1|M],[1,5|N],O) .

addlists_1([],[],[]).
addlists_1([A|B],[C|D],[E|F]) :-
     E is A+C, addlists_1(B,D,F) .

addlists_5([A,A],[_1,_2],[4,7]).
addlists_5([A,A,B|C],[_1,_2,D|E],[4,7,F|G]) :-
     F is B+D, addlists_1(C,E,G).

addlists_6([7,1],[1,5],[8,6]).
addlists_6([7,1,A|B],[1,5,C|D],[8,6,E|F]) :-
     E is A+C, addlists_1(B,D,F).
```

Figure 4.3: Specialization of `addlists/3` after Minimization.

4.2 A General View of Polyvariance and Minimization

We now present a very general description of a polyvariant specialization process which includes both partial evaluation [85, 44, 71] and abstract multiple specialization [108].

Given a program P and a set of atoms $\mathcal{A} = \{A_1, \ldots, A_m\}$, which describe the possible initial queries to P, polyvariant specialization performs the following three steps:

1. *Analysis.* In this phase, we compute a set of call patterns $\{A_1, \ldots, A_n\} \supseteq \mathcal{A}$ which cover all calls in the specialized program. We write $Analysis(P, \mathcal{A}) = \{A_1, \ldots, A_n\}$ to denote that the result of analysis for P and $\mathcal{A}$ is the set of call patterns $\{A_1, \ldots, A_n\}$.

2. *Code Generation.* The aim of this phase is, for each call pattern $A_i \in Analysis(P, \mathcal{A})$, to compute properly optimized residual code. We denote by $code(A_i)$ the code (set of clauses) associated to A_i. In partial evaluation, an unfolding rule U is used for generating code, i.e.,

$$code(A_i) = U(P, A_i).$$

3. *Renaming.* In this phase we assign a fresh predicate name to each atom in $\{A_1, \ldots, A_n\}$. Then, for each $code(A_i)$, we perform appropriate renamings in the head and body atoms so that each program point uses a correct (and as optimized as possible) version. *Ren* denotes the renaming function.

The polyvariant specialized program $P_{\mathcal{A}}$ is then defined as:

$$P_{\mathcal{A}} = \bigcup_{A_i}^{Analysis(P,\mathcal{A})} Ren(code(A_i))$$

4.2.1 Minimizing the Results of Polyvariant Specialization

The aim of minimization is to group the call patterns (or *versions*) in $\{A_1, \ldots, A_n\}$ into *equivalence classes*, obtaining a minimal program that allows the same set of

optimizations, and that can be implemented without introducing run-time tests to select amongst different versions of a predicate.

We start recalling some definitions introduced in [109].

Definition 4.2.1 (feasible program)**.** *Is a specialized program in which no run-time tests are introduced.*

Definition 4.2.2 (minimal program)**.** *A specialized program is* minimal *if whenever two call patterns are equivalent, they are placed into the same version.*

Definition 4.2.3 (program of maximal optimization)**.** *We say that a program is of* maximal optimization *if no two call patterns with different optimizations are placed into the same version.*

Using this terminology, we can rephrase our earlier statement to say that the goal of minimization is to group call patterns in $\{A_1, \ldots, A_n\}$ into *equivalence classes* in order to obtain a feasible minimal program of maximal optimization.

Deciding whether two versions A_i and A_j with $pred(A_i) = pred(A_j)$ are equivalent is not straightforward, as we have to consider not only the code of A_i and A_j, but also the code of all other versions which are reachable from them. In the case of the *main* predicate in a program, we would have to take the code of all the specialized program into account. Thus, we will split the notion of equivalence into a *local equivalence* and a *global equivalence* level.

Local equivalence Local equivalence concentrates on comparing the code for A_i and A_j only, without worrying about the other predicates which are reachable from them.

Global equivalence Global equivalence will only hold if A_i and A_j are locally equivalent and all reachable versions for the corresponding program points are also locally equivalent.

The minimization algorithm (called *Minimize* from now on) consists of two phases.

Reunion phase the first phase is called *reunion* and its aim is to obtain a program of maximal optimization while remaining minimal. This is done by producing a partition $\{V_1, \ldots, V_m\}$ from a given a set of atoms $\{A_1, \ldots, A_n\}$,

with $m \leq n$ s.t. $\forall A, A' \in V_i \ . \ A \equiv A'$, where $A \equiv A'$ denotes the fact that A and A' are *locally equivalent*.

Unfortunately, if we generate code from versions resulting from this phase, we would obtain a program which is not feasible in general. This is because two call patterns allowing the same set of local optimizations may use different versions for the same literal, and thus they cannot be blindly collapsed into the same version.

Splitting phase the second phase is called *splitting*, and its aim is to obtain a feasible program, while remaining minimal and of maximal optimization. Further details about this algorithm can be found in [109].

The reunion phase is concerned with local equivalence only and it places together all versions for the same predicate which are considered locally equivalent according to some criteria. The splitting phase is concerned with global equivalence and splits sets of versions which are not globally equivalent until no more splitting is needed, i.e., until we have reached a partition where all sets contain versions which are globally equivalent.

This minimization process is isomorphic to the minimization of deterministic finite automata (DFA) [57], by considering each call pattern A_i as a *state* and each program point in $code(A_i)$ as a *symbol*. In [57] an algorithm is proposed for, given a DFA M, achieving a minimal DFA M' equivalent to M. If M has k symbols and n states, the complexity of this algorithm is $O(kn^2)$. The algorithm consists of two phases.

- In the first one, pairs of states (atoms) which are candidates for being equivalent are identified. All other pairs are marked as not equivalent.
- Then, the second phase keeps on marking pairs of states which are not equivalent until all pairs of potentially equivalent states are visited. Two states (call patterns) are not equivalent when they behave differently for the same symbol (program point), i.e., they call predicates which have been identified not to be equivalent.

A crucial point thus is, given a pair of atoms A and A', to decide whether they can be safely considered locally equivalent. The decision criteria has to satisfy two properties:

1. it must produce correct results, and

2. it must be effective, i.e. it must be possible to efficiently decide whether A and A' are candidates for equivalence based on syntactic, local conditions.

For this purpose, we use the notion of *structural equivalence*.

Definition 4.2.4 (structurally equivalent)**.** *Let A_1 and A_2 be two call patterns such that $pred(A_1) = pred(A_2)$. We say that A_1 and A_2 are* structurally equivalent *iff*

$$\begin{aligned} & C = msg(code(A_1), code(A_2)) \\ \wedge\ & instantiate(C, A_1) \approx code(A_1) \\ \wedge\ & instantiate(C, A_2) \approx code(A_2) \end{aligned}$$

where $A_1 \approx A_2$ denotes that A_1 and A_2 are *variants*, as mentioned in Chapter 2. Clearly, if $code(A_1) \approx code(A_2)$ then A_1 and A_2 are structurally equivalent. However the definition above allows also considering as structurally equivalent call patterns whose code only differs in constants which are input arguments to the predicate but which do not play an important role for local optimization.

Note that structural equivalence is just a syntactic characterization which guarantees that two call patterns are locally equivalent. In fact, there can be call patterns which are locally equivalent in the sense that their behaviours under the semantics of interest are identical but which our definition of structural equivalence would not capture.

Also, structural equivalence in particular, and local equivalence in general do not guarantee global equivalence. It often happens that two call patterns which are structurally equivalent end up in different equivalence classes after the splitting phase. Only after this phase terminates we can be sure that two call patterns are globally equivalent.

The polyvariant specialized program with minimization $P_{\mathcal{A}}^{Min}$ is defined as:

$$P_{\mathcal{A}}^{Min} = \bigcup_{V_i}^{Minimize(Analysis(P,\mathcal{A}))} Ren_{\equiv}(code(V_i))$$

where given a set of atoms $\{A_1, \ldots, A_n\}$, we partition them in equivalence classes $\{V_1, \ldots, V_k\}$, $k \leq n$, s.t. $\forall A, A' \in V_i$. A and A' are structurally equivalent. We

use $code(\{A_1, \ldots, A_i\})$ to denote $msg(\{code(A_1), \ldots, code(A_i)\})$. Also, $Ren_{\equiv}$ is a new renaming function which always uses the same (*canonical*) predicate name for any atom in $\{A_1, \ldots, A_i\}$.

Our definition of structural equivalence plays several roles. It underlies the notions of local equivalence used both in abstract multiple specialization and partial evaluation, thus allowing us to present a unified view of both minimization processes. Furthermore, it can also be used in order to determine whether two versions are locally equivalent.

Existing approaches to minimization do not compare the syntactic structure of the residual code directly (as this definition would require) but rather use the specialization history in order to decide local equivalence. In [108] two call patterns are considered locally equivalent iff (1) they correspond to the same predicate in the original program and (2) the set of optimizations in both call patterns is the same. In [84] two call patterns are locally equivalent iff they have the same characteristic tree.

In a way, we could think that given two call patterns A_1 and A_2, the task of checking whether they are structurally equivalent may be done in one of the following scenarios:

- using *no additional information* at all, by just applying the definition of structural equivalence to A_1 and A_2, i.e., generating code from the call patterns, obtaining their *msg*, and then instantiating back using A_1 and A_2 to determine whether a variant of the original code is obtained. Clearly, even though this is a feasible approach, it may also be very inefficient.

- using *syntactic information*, such as determining that A_1 and A_2 are candidates for being structurally equivalent if they have the same original predicate name, and when the amount of clauses of $code(A_1)$ and $code(A_2)$ match. When these conditions are met, then we can check whether A_1 and A_2 are structurally equivalent as before. This method is clearly more efficient than using no information at all.

- using *specialization history*. We could also determine that any two call patterns are candidates for minimization if their specialization history is the same. Their specialization history can be abstracted away by means of their characteristic trees [42, 84] or trace terms [43]. Under the usual assumption

that the unfolding strategy must perform at least one unfolding step, the fact that two call patterns have the same characteristic tree (or trace term) implies that they correspond to the same predicate in the original program. This will be explained in detail in the next three sections.

4.3 Characteristic Trees with External Predicates

Characteristic trees were introduced in [42] and also used in [78, 84]. Their aim is to capture all the relevant aspects of the unfolding process.

A *characteristic tree* is a data structure which encapsulates the evaluation behaviour of an atom, i.e., a trace of the unfolding process. This provides a powerful mechanism to guide generalization and polyvariance throughout the transformation process.

The following definitions are taken from [84], which in turn were derived from [42], and the SP system [40]).

Definition 4.3.1 (characteristic path)**.** *Let G_0 be a goal, and let P be a definite program whose clauses are numbered. Let $G_0, \ldots, G_n$ be the goals of a finite, possibly incomplete SLD-derivation D of $P \cup \{G_0\}$. The characteristic path of the derivation D is the sequence $\langle l_0 : c_0, \ldots, l_{n-1} : c_{n-1} \rangle$, where l_i is the position of the selected atom in G_i, and c_i is the number of the clause chosen to resolve with G_i.*

Now that we have characterized derivations, we can characterize goals through the derivations in their associated SLD-trees.

Definition 4.3.2 (characteristic tree)**.** *Let G be a goal, P a definite program, and τ a finite SLD-tree for $P \cup \{G\}$. Then the characteristic tree $\hat{\tau}$ of τ is the set containing the characteristic paths of the nonfailing SLD-derivations associated with the branches of τ.*
Let U be an unfolding rule such that $U(P, G) = \tau$. Then $\hat{\tau}$ is also called the characteristic tree of G (in P) via U. We introduce the notation ch_tree$(G, P, U) = \hat{\tau}$.

Example 4.3.3. *Let P be the* `append` *program:*

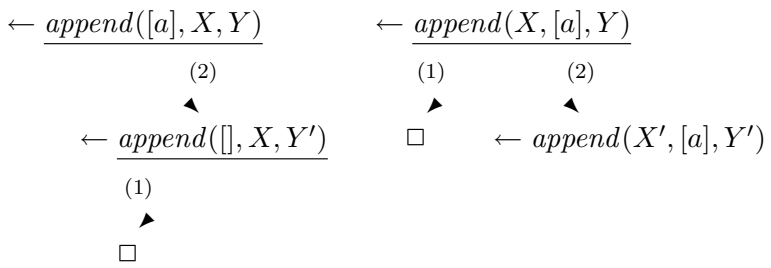

Figure 4.4: SLD-trees τ_A and τ_B for Example 4.3.3.

```
(1) append([],Z,Z).
(2) append([H|X],Y,[H|Z]) :- append(X,Y,Z).
```

We have added clause numbers, which are incorporated into Figure 4.4 in order to clarify which clauses have been resolved with. To avoid cluttering this figure we have dropped the substitutions.

Given for example the atom A=`append([a],X,Y)`*, we can fully unfold it during partial evaluation, obtaning the resultant*

```
append([a],X,[a|X]).
```

The corresponding SLD-tree τ_A *is depicted in Figure 4.4, and its associated characteristic tree is* $\{\langle 1:2, 1:1\rangle\}$.

If we take the atom B=`append(X,[a],Y)` *and perform partial evaluation, a possible outcome can be the set of clauses*

```
append([],[a],[a]).
append([H|X],[a],[H|Z]) :- append(X,[a],Z).
```

The corresponding SLD-tree τ_B *is depicted in Figure 4.4, and its associated characteristic tree is* $\{\langle 1:1\rangle, \langle 1:2\rangle\}$.

4.3.1 Handling Builtins in Characteristic Trees

Although existing partial evaluators such as SP [40] and ECCE [79] perform some limited handling of builtins within characteristic trees, the existing formal definitions of characteristic trees do not contemplate the existence of builtins nor of external predicates. We now extend the standard definitions in order to

accurately include external predicates. This will allow us to introduce powerful sufficient conditions for isomorphism of characteristic trees in Section 4.4 below.

Definition 4.3.4 (characteristic path with external predicates)**.** *Let G_0 be a goal, and let P be a program whose clauses are numbered. Let $G_0, \ldots, G_n$ be the goals of a finite, possibly incomplete SLD-derivation D of $P \cup \{G_0\}$. Let $A_0, \ldots, A_{n-1}$ be the selected atoms in D. The* characteristic path with external predicates *of the derivation D is the sequence $\langle l_0 : c_0, \ldots, l_{n-1} : c_{n-1} \rangle$, where l_i is the position of A_i in G_i, and c_i is defined as follows:*

- *if $pred(A_i)$ is defined in P, then c_i is the number of the clause in P chosen to resolve with G_i;*
- *if $pred(A_i)$ is an external predicate, then let θ be a computed answer generated when performing $exec(A_i)$. Then, c_i is a pair (A_i, θ).*

In the definition above, $exec(A_i)$ represents the execution of A_i. For this, the external call A_i has to be *evaluable* [106], i.e., A_i is both well-moded and well-typed, it does not produce any side-effect, and it universally terminates. Note that $exec(A_i)$ can succeed more than once and possibly with different computed answers.

Reconsidering characteristic paths, each pair $(l_i : c_i)$ in a characteristic path must uniquely identify:

1. the position of the selected atom A_i,
2. the bindings introduced by this step on the current goal, and
3. the atoms which must be introduced in the goal in place of the selected atom A_i.

An important obvious difference between external and regular predicates is that the code for external predicates may not be available, so it is not possible, as done with regular predicates, to assign clause numbers to them or to unfold them. Instead of unfolding external predicates, we will fully execute them. As a result, no atoms will be introduced in the current goal and, thus, (3) is not needed in this case.

In the case of external predicates, we introduce in the characteristic tree an *external success*, i.e., a pair (A_i, θ) containing the call pattern A_i and the bindings θ generated during evaluation for each external predicate. Note that, in contrast to the handling of builtins within characteristic trees in the systems SP and ECCE, this makes it possible to reconstruct the residual code for an atom without the need for (re-)evaluating external predicates, even if the external predicates succeed several times with (possibly) different computed answers. The notion of characteristic paths with external predicates is indeed consistent with traditional characteristic paths. In the case of regular predicates, the same *implicit* representation as in traditional characteristic paths is used. This representation is efficient in space since rather than introducing (an instantiated version of) the clause chosen for resolving the selected atom directly in the characteristic tree, only the number of the clause used for unfolding is stored. This suffices since the actual instantiation can be performed later if needed using the actual clause. In the case of external predicates, this implicit representation is no longer possible, since the clauses are not available. Instead, the call pattern and the corresponding bindings are *explicitly* stored.

Definition 4.3.5 (characteristic trees with external predicates)**.** *Let G be a goal, P a definite program, and τ a finite SLD-tree for $P \cup \{G\}$. Then the* characteristic tree with external predicates *of τ is the set containing the characteristic paths with external predicates of the non-failing SLD-derivations associated with the branches of τ. We also assume from now on that "ch_tree" refers to characteristic trees with external predicates.*

Let U be an unfolding rule such that $U(P, G) = \tau$. Then $\hat{}\tau$ is also called the characteristic tree of G (in P) via U, and we use the notation ch_tree$(G, P, U) = \hat{}\tau$ to denote it.

Characteristic trees are extended to handle external predicates by simply considering characteristic paths with external predicates. Figure 4.5 shows the characteristic trees with external predicates τ_2, τ_3, τ_4, τ_5 and τ_6 for versions `addlists_2/3`, `addlists_3/3`, `addlists_4/3`, `addlists_5/3`, and `addlists_6/3`, respectively.

$$\begin{aligned}
\tau_2 = \ & \{\langle 1:3, 1:(4\ is\ 4+0, \epsilon), 1:3, 1:(7\ is\ 4+3, \epsilon), 1:2\rangle, \\
& \ \langle 1:3, 1:(4\ is\ 4+0, \epsilon), 1:3, 1:(7\ is\ 4+3, \epsilon), 1:3\rangle\}, \\
\tau_3 = \ & \{\langle 1:3, 1:(4\ is\ 3+1, \epsilon), 1:3, 1:(7\ is\ 3+4, \epsilon), 1:2\rangle, \\
& \ \langle 1:3, 1:(4\ is\ 3+1, \epsilon), 1:3, 1:(7\ is\ 3+4, \epsilon), 1:3\rangle\}, \\
\tau_4 = \ & \{\langle 1:3, 1:(A\ is\ 3+1, \{A \mapsto 4\}), 1:3, 1:(B\ is\ 3+4, \{B \mapsto 7\}), 1:2\rangle, \\
& \ \langle 1:3, 1:(C\ is\ 3+1, \{C \mapsto 4\}), 1:3, 1:(D\ is\ 3+4, \{D \mapsto 7\}), 1:3\rangle\}, \\
\tau_5 = \ & \{\langle 1:3, 1:(E\ is\ 1+3, \{E \mapsto 4\}), 1:3, 1:(F\ is\ 1+6, \{F \mapsto 7\}), 1:2\rangle, \\
& \ \langle 1:3, 1:(G\ is\ 1+3, \{G \mapsto 4\}), 1:3, 1:(H\ is\ 1+6, \{H \mapsto 7\}), 1:3\rangle\}, \\
\tau_6 = \ & \{\langle 1:3, 1:(I\ is\ 7+1, \{I \mapsto 8\}), 1:3, 1:(J\ is\ 1+5, \{J \mapsto 6\}), 1:2\rangle, \\
& \ \langle 1:3, 1:(L\ is\ 7+1, \{L \mapsto 8\}), 1:3, 1:(M\ is\ 1+5, \{M \mapsto 6\}), 1:3\rangle\}.
\end{aligned}$$

Figure 4.5: Characteristic Trees for `addlists/3` Versions.

4.4 Isomorphic Characteristic Trees

In this section we provide novel sufficient conditions for considering two call patterns as locally equivalent. For this we define the notion of *isomorphic* characteristic trees with external predicates, which guarantees that the corresponding code is structurally equivalent. We assume that predicate names cannot be numbers, as is the case in most existing logic programming systems. Also, *number(X)* succeeds iff X is a number.

First, we introduce the concept of *quasi-isomorphic characteristic trees*, for identifying characteristic trees which only (possibly) differ in the input and/or output values of arguments in calls to external predicates:

Definition 4.4.1 (quasi-isomorphic characteristic trees)**.** *Two characteristic paths* $\delta^1 = \langle l_0 : c_0^1, \ldots, l_m : c_m^1\rangle$ *and* $\delta^2 = \langle l_0 : c_0^2, \ldots, l_m : c_m^2\rangle$ *are* quasi-isomorphic *and we denote it* $\delta^1 \approx_q \delta^2$ *iff* $\forall i \in \{1..m\}\ .\ number(c_i^1) \Rightarrow c_i^1 = c_i^2$.

Two characteristic trees τ_1 *and* τ_2 *are* quasi-isomorphic*, denoted* $\tau_1 \approx_q \tau_2$*, iff*

- $\forall \delta^1 \in \tau_1\ .\ \exists \delta^2 \in \tau_2\ s.t.\ \delta^1 \approx_q \delta^2$ *and*
- $\forall \delta^2 \in \tau_2\ .\ \exists \delta^1 \in \tau_1\ s.t.\ \delta^2 \approx_q \delta^1$.

Note that quasi-isomorphic characteristic paths must have the same length and the selected atom must be in the same position in each resolution step. Further-

more, if the atom is not for an external predicate, then the atom must have been resolved against the same clause. In Figure 4.5, $\tau_2 \approx_q \tau_3 \approx_q \tau_4 \approx_q \tau_5 \approx_q \tau_6$.

Now we define some relationships among external successes, after some auxiliary definitions. A *position* uniquely determines a subterm within a term.

Definition 4.4.2 (position)**.** *A* position ω *is either the empty position* ε*, or* $n.\omega'$*, where* n *is a natural number and* ω' *is a position.*

Definition 4.4.3 (getval, Pos, and Allpos)**.** *Let* $A = f(\overline{t_n})$ *be a term. Let* ω *be a position. Let* X *be a variable s.t.* $X \in vars(A)$*. Let* θ *be a substitution.*

- *We define* $getval(\omega, A)$ *as* A *if* $\omega = \varepsilon$ *and as* $getval(\omega', t_i)$ *if* $\omega = i.\omega'$*.*
- *We define* $Pos(A, X)$ *as* $\{\omega \mid getval(\omega, A) = X\}$*.*
- *We define* $Allpos(A, \theta)$ *as* $\cup_{X \in dom(\theta)}\{\omega\}$*, s.t.* $\omega \in Pos(A, X)$*.*

Example 4.4.4. $getval(2.1.\varepsilon, f(a, g(b, c))) = b$*, and* $Pos(f(a, g(b, Y)), Y) = \{2.2.\varepsilon\}$*. If* A *is not linear, then for some* X*, the set* $Pos(A, X)$ *may have more than one element. E.g.,* $Pos(f(Z, g(Z)), Z) = \{1.\varepsilon, 2.1.\varepsilon\}$*. In such case, any* $\omega \in Pos(A, X)$ *can be used for our purposes. Also* $Allpos(A \text{ is } 3 + 1, \{A \mapsto 4\}) = \{1.\varepsilon\}$*.*

Definition 4.4.5 (isomorphic external successes)**.** *Let* $c = (A, \theta)$ *and* $c' = (A', \theta')$ *be external successes. Then* c *and* c' *are* isomorphic external successes*, denoted by* $c \simeq c'$*, iff* $\forall \omega \in Allpos(A, \theta) \cup Allpos(A', \theta')$ *.* $getval(\omega, A\theta) = getval(\omega, A'\theta')$*.*

Example 4.4.6. *This definition tries to consider as isomorphic as many pairs of external successes as possible. A particular subcase of this definition corresponds to the case where the calls to external predicates generate no bindings. For example, the pair* $(4 \text{ is } 4 + 0, \epsilon)$ *and* $(4 \text{ is } 3 + 1, \epsilon)$ *is isomorphic, whereas the notion of equivalence in [84] cannot capture this since the builtin predicate* `is/2` *potentially generates bindings, though in this case it does not. Note that* $(4 \text{ is } 4 + 0, \epsilon)$ *and* $(8 \text{ is } 2 * 4, \epsilon)$ *are also considered as isomorphic although their syntactic structure is very different.*

Another interesting subcase is when the external successes have different levels of instantiation but on success they are variants. This happens with $(A \text{ is } 3 + 1, \{A \mapsto 4\})$ *and* $(4 \text{ is } 3 + 1, \epsilon)$*, that are isomorphic according to Definition 4.4.5.*

Furthermore, it allows considering as isomorphic external successes which have the same values in all positions which are instantiated in either external success. For example $(A \ is \ 3+1, \{A \mapsto 4\})$ *and* $(4 \ is \ 4+0, \epsilon)$ *are considered isomorphic since*

- $Allpos(A \ is \ 3+1, \{A \mapsto 4\}) = \{1.\varepsilon\} \wedge$
- $Allpos(4 \ is \ 4+0, \epsilon\}) = \emptyset \wedge$
- $getval(1.\varepsilon, 4 \ is \ 3+1) = getval(1.\varepsilon, 4 \ is \ 4+0) = 4$

However, $(E \ is \ 1+3, \{E \mapsto 4\}) \not\simeq (I \ is \ 7+1, \{I \mapsto 8\})$*, since*

- $Allpos(E \ is \ 1+3, \{E \mapsto 4\}) = Allpos(I \ is \ 7+1, I) = \{1.\varepsilon\}$*, but*
- $getval(1.\varepsilon, 4 \ is \ 1+3) = 4 \neq getval(1.\varepsilon, 8 \ is \ 7+1) = 8$.

As a side note, the minimization approach in [84] only considers as isomorphic a restricted version of $\simeq_\epsilon$ where the external predicate involved in c and c' must be well-known and safe. In such approach, none of the external successes above are considered isomorphic since predicate `is/2` is not safe in general, as it can generate bindings for its first argument.

Definition 4.4.7 (isomorphic characteristic trees)**.** *Two characteristic paths* $\delta^1 = \langle l_0 : c_0^1, \ldots, l_m : c_m^1 \rangle$ *and* $\delta^2 = \langle l_0 : c_0^2, \ldots, l_m : c_m^2 \rangle$ *are* isomorphic *and we denote it* $\delta^1 \approx \delta^2$ *iff*

- $\delta^1 \approx_q \delta^2$ *and*
- $\delta^1 \approx_q \delta^2 \wedge \forall i \in \{1..m\} \ . \ c_i^1 = (A_i^1, \theta_i^1) \Rightarrow c_i^2 = (A_i^2, \theta_i^2) \wedge c_i^1 \simeq c_i^2$.

Two characteristic trees τ_1 *and* τ_2 *are* isomorphic*, denoted* $\tau_1 \approx \tau_2$*, iff* $\forall \delta^1 \in \tau_1 \ . \ \exists \delta^2 \in \tau_2$ *s.t.* $\delta^1 \approx \delta^2$ *and* $\forall \delta^2 \in \tau_2 \ . \ \exists \delta^1 \in \tau_1$ *s.t.* $\delta^2 \approx \delta^1$.

The following proposition provides the basis for our minimization approach.

Proposition 4.4.8 (structural equivalence)**.** *Let* P *be a program with external predicates, let* U *be an unfolding rule, let* A_1 *and* A_2 *be two call patterns such that* $\tau_1 = ch\_tree(A_1, P, U)$ *and* $\tau_2 = ch\_tree(A_2, P, U)$*. If* $\tau_1 \approx \tau_2$ *then* A_1 *and* A_2 *are structurally equivalent.*

A difficulty with our notion $\approx$ of isomorphic characteristic trees and its usage as a condition for local equivalence is that though the $\approx$ relation is reflexive and symmetric, it is not transitive. This means that $(\tau_1 \approx \tau_2 \wedge \tau_2 \approx \tau_3) \not\rightarrow \tau_1 \approx \tau_3$. As a result, in order to be able to state that all characteristic trees in a set $\{\tau_1, \ldots, \tau_n\}$ are isomorphic we have to check that $\forall \tau, \tau' \in \{\tau_1, \ldots, \tau_n\}\ .\tau \approx \tau'$.

Example 4.4.9. *Let us consider again the characteristic trees in Figure 4.5. We have already noticed that all of them are quasi-isomorphic. If we take the quasi-isomorphic paths of τ_2, τ_3, τ_4 and τ_5, and extract their external successes, we can see that they are isomorphic. For example, if we take $c_{21} = (4\ is\ 4+0, \epsilon)$, $c_{31} = (4\ is\ 3+1, \epsilon)$, $c_{41} = (A\ is\ 3+1, \{A \mapsto 4\})$ and $c_{51} = (C\ is\ 1+3, \{C \mapsto 4\})$, we can compute $\cup_{i \in \{2 \ldots 5\}} Allpos(c_{i1}) = \{1.\epsilon\}$. Since $getval(1.\epsilon, 4\ is\ 4+0) = getval(1.\epsilon, 4\ is\ 3+1) = getval(1.\epsilon, 4\ is\ 1+3) = 4$, we can conlude that they are isomorphic.*

However, note that even though $\tau_5 \approx_q \tau_6$, they are not (fully) isomorphic since, for instance, $(E\ is\ 1+3, \{E \mapsto 4\}) \not\approx (I\ is\ 7+1, \{I \mapsto 8\})$. Indeed, `addlists_5/3` *and* `addlists_6/3` *are not structurally equivalent.*

By Prop. 4.4.8 the sets which are identified as locally equivalent during the reunion phase are:

- {`main/15`},
- {`addlists_1/3`},
- {`addlists_2/3`,`addlists_3/3`,`addlists_4/3`,`addlists_5/3`},
- {`addlists_6/3`}.

This is also the final partition after applying the splitting phase. This produces the minimized program which was shown in Figure 4.3.

In the implementation, in order to reduce the cost of checking every characteristic tree against all other trees which are quasi-isomorphic to it before being able to consider them isomorphic, each set of versions is augmented with a canonical representative, which stores the set of positions in each of the members in the set. This way, a new candidate only needs to be checked against such canonical representative but taking into account all positions stored for them, and not only the ones which actually are bound at the corresponding external successes.

4.5 Local Trace Terms

Trace terms [43] are an abstraction of AND-trees, which are a representation of *successful* computation paths. They are similar in some ways to characteristic trees, the main difference being that trace terms abstract away the computation rule, and that they represent complete answers rather than partial unfolding traces.

In this section we introduce *local* trace terms, which extend trace terms allowing to reflect traces of incomplete derivations. We start by introducing first clause identifiers.

Definition 4.5.1 (clause identifiers)**.** *Let P be a normal program, and $\{c_1 \dots c_n\}$ the set of clauses in P. Let $a_i, 1 \leq i \leq n$ be the number of atoms in the body of c_i . Then each clause c_i in P is associated with a functor φ_i/a_i, where φ_i is not in the language of P, and $\varphi_i/a_i \neq \varphi_j/a_j$ iff $i \neq j$. These functors are called* clause identifiers.

Any successful SLD-derivation can be transformed into an AND-tree. Now we introduce an extension to the concept of AND-trees, in order to support incomplete computations. Thus, an *incomplete* AND-tree can represent either a successful computation or an incomplete one.

Definition 4.5.2 (incomplete AND-trees)**.** *Let P be a program. Then an extended AND-tree for P is a tree where each node can be either a*

non-leaf node *labelled by a clause $A \leftarrow A_1, \dots, A_k$, and an atom $A\theta$ (for some substitution θ), and has children $A_1\theta, \dots, A_k\theta$, or*

leaf node *which can be further classified in*

- **final** *labelled by a clause $A \leftarrow true$ and an atom θ, for some substitution θ,*
- **local** *labelled by a clause $A \leftarrow A_1, \dots, A_k$, and an atom $A\theta$ (for some substitution θ), representing an incomplete computation and therefore having no children, and*
- **external** *labelled by an external success (B, θ') and an atom $B\theta$, where B is an external predicate, θ' is the computed answer generated when*

$$\alpha_2 = addl2((4\ is\ 4+0, \epsilon), addl2((7\ is\ 4+3, \epsilon), X_2))$$

$$\alpha_3 = addl2((4\ is\ 3+1, \epsilon), addl2((7\ is\ 3+4, \epsilon), X_3))$$

$$\alpha_4 = addl2((A\ is\ 3+1, \{A \mapsto 4\}), addl2((B\ is\ 3+4, \{B \mapsto 7\}), X_4))$$

$$\alpha_5 = addl2((C\ is\ 1+3, \{C \mapsto 4\}), addl2((D\ is\ 1+6, \{D \mapsto 7\}), X_5))$$

$$\alpha_6 = addl2((E\ is\ 7+1, \{E \mapsto 8\}), addl2((F\ is\ 1+5, \{F \mapsto 6\}), X_6))$$

Figure 4.6: Local Trace Terms for `addlists/3` Versions.

executing B, and θ is some substitution. Here θ' can the empty substitution ϵ if B was not executed.

We now introduce *local* trace terms, which abstract incomplete AND-trees.

Definition 4.5.3 (local trace terms)**.** *Let T be an AND-tree, then a local trace term for T, denoted by $\alpha(T)$, is either*

- *φ_i, if T is a* final *leaf node labelled by the clause identified by φ_i/a_i, or*
- *X, if T is a* local *leaf node, where $X \in Vars$ is an arbitrary variable, or*
- *(B, θ), if T is an* external *leaf node labelled by (B, θ), or*
- *$\varphi_i(\alpha(T_1), \ldots, \alpha(T_{a_i}))$, if T is labelled by φ_i/a_i and has immediate subtrees $T_1, \ldots, T_{a_i}$.*

Let P be a program, and $\leftarrow A$ be a goal. Let T be an AND-tree for P with root labelled by $A\theta$, and α a local trace term abstracting T. Then we introduce the notation $ltt(P, A) = \alpha$.

For example, given the program in Figure 4.1, where functors `m/6,addl1/0,addl2/2` are assigned to clauses 1, 2 and 3 respectively, we show in Figure 4.6 the local trace terms α_2, α_3, α_4, α_5 and α_6 for versions `addlists_2/3`, `addlists_3/3`, `addlists_4/3`, `addlists_5/3`, and `addlists_6/3`, respectively.

Intuitively, if two call patterns have *similar enough* associated trace terms, then we can consider them as structurally equivalent. This notion of *similarity* is formalized in terms of isomorphism, as we did above for characteristic trees with external predicates.

Definition 4.5.4 (isomorphic local trace terms)**.** *Two local trace terms* α^1 *and* α^2 *are isomorphic, denoted by* $\alpha^1 \approx \alpha^2$*, iff*

- $\alpha^1 = \alpha^2 = \varphi$*, i.e, they are the same clause identifier, or*
- $\alpha^1 \wedge \alpha^2$ *are both variables, or*
- $\alpha^1 \wedge \alpha^2$ *are isomorphic external successes, i.e.* $\alpha^1 \approx \alpha^2$ *(by Def. 4.4.5), or*
- $\alpha^1 = \varphi(\alpha^1_1, \ldots, \alpha^1_n) \wedge \alpha^2 = \varphi(\alpha^2_1, \ldots, \alpha^2_n)$*, and* $\forall i \in \{1 \ldots n\}.\alpha^1_i \approx \alpha^2_i$*.*

The following proposition formalizes the notion that when two call patterns have isomorphic local trace terms, then they are structurally equivalent.

Proposition 4.5.5 (structural equivalence)**.** *Let* P *be a program with external predicates, let* A_1 *and* A_2 *be two call patterns such that* $\alpha_1 = ltt(P, A_1)$ *and* $\alpha_2 = ltt(P, A_2)$*. If* $\alpha_1 \approx \alpha_2$ *then* A_1 *and* A_2 *are structurally equivalent.*

Let us consider the local trace terms in Fig. 4.6. If we take, for example, α_2 and α_3, it can be easily verified that they share the same clause identifiers, and that their external successes are isomorphic (already shown in Sec. 4.4), therefore by Prop. 4.5.5 we can consider versions `addlists_2/3` and `addlists_3/3` as structurally equivalent. It is not difficult to follow the same reasoning and reach the same results as in Sec. 4.4, obtaining as candidates for minimization the sets

- {`main/15`},
- {`addlists_1/3`},
- {`addlists_2/3`,`addlists_3/3`,`addlists_4/3`,`addlists_5/3`},
- {`addlists_6/3`}.

4.6 Minimization via Residualization of External Calls

In previous sections we have established that call patterns with isomorphic characteristic trees (or isomorphic local trace terms) are structurally equivalent, and

therefore they can be collapsed into the same version. This makes sense if we want to have a program of *maximal optimization*. However, there are situations in which even the minimized program is too large and/or where we would like to trade space for time efficiency. This would mean achieving programs which are smaller, but at the cost of introducing some efficiency penalty. In cases like this, we propose as candidates for minimization, call patterns with *quasi-isomorphic* characteristic trees.

An important observation is that if $\delta^1 \approx_q \delta^2$ then the associated resultants have the same structure. However, this is not a sufficient condition for structural equivalence. This is because part of the bindings needed for structural equivalence cannot be achieved by the operation *instantiate*, as in Def. 4.2.4, but rather they originate from the execution of calls to external predicates. Thus, the second important observation is that if the calls to external predicates involved succeed only once, i.e. they are deterministic, such missing bindings can be recovered at run-time by residualizing (part of the) calls to external predicates which had in principle taken place during specialization time.

Note that for detecting determinacy, no static analysis is actually required. We can simply check whether the calls which are to be residualized succeed just once by directly executing the calls as they appear in the different characteristic trees, i.e., before applying the *msg* to them. After the required external predicates have been residualized, the corresponding versions will be structurally equivalent. The strategy we propose is the following: for any pair of versions A_1 and A_2 with $\tau_1 = ch\_tree(A_1, P, U)$ and $\tau_2 = ch\_tree(A_2, P, U)$ s.t. $\tau_1 \approx_q \tau_2$ we:

1. Let $(A, \theta) \in \delta \in \tau_i,\ i \in \{1..2\}$, then we replace it by $A\theta$, i.e., we apply the corresponding substitution θ to each external success $A\theta$.

2. Compute $(C, T) = msg((code(A_1), \overline{\tau}_1), (code(A_2), \overline{\tau}_2))$, where $\forall i \in \{1..2\}.\overline{\tau}_i$ is obtained from τ_i by *evaluating* all external successes, i.e., $\forall (B, \theta)$ we replace it by $B\theta$. Since $\tau_1 \approx_q \tau_2$, we can simply compute the *msg* of the *evaluated* external successes, i.e, given (A, θ) we apply the *msg* to $A\theta$, instead of using the whole tree.

3. **If** $\forall i \in \{1..2\}$. $instantiate(C, A_i) \approx code(A_i)$

   - **then** A_1 and A_2 are structurally equivalent. No need to residualize.

$$msg\left(\begin{array}{l}\{addlists([4,4],[0,3],[4,7]).,\langle 1:(4\ is\ 4+0),1:(7\ is\ 4+3)\rangle\}\\ \{addlists([3,3],[1,4],[4,7]).,\langle 1:(4\ is\ 3+1),1:(7\ is\ 3+4)\rangle\}\\ \{addlists([3,3],[1,4],[4,7]).,\langle 1:(4\ is\ 3+1),1:(7\ is\ 3+4)\rangle\}\\ \{addlists([1,1],[3,6],[4,7]).,\langle 1:(4\ is\ 1+3),1:(7\ is\ 1+6)\rangle\}\end{array}\right)$$
$$\{addlists([X,X],[Y,Z],[4,7]).,\langle 1:(4\ is\ X+Y),1:(7\ is\ X+Z)\rangle\}$$

$$msg\left(\begin{array}{l}\{addlists([4,4,A|B],[0,3,C|D],[4,7,E|F]):-E\ is\ A+C,addlists(B,D,F).,\\ \qquad\langle 1:(4\ is\ 4+0),1:(7\ is\ 4+3)\rangle\}\\ \{addlists([3,3,A|B],[1,4,C|D],[4,7,E|F]):-E\ is\ A+C,addlists(B,D,F).,\\ \qquad\langle 1:(4\ is\ 3+1),1:(7\ is\ 3+4)\rangle\}\\ \{addlists([3,3,A|B],[1,4,C|D],[4,7,E|F]):-E\ is\ A+C,addlists(B,D,F).,\\ \qquad\langle 1:(4\ is\ 3+1),1:(7\ is\ 3+4)\rangle\}\\ \{addlists([1,1,A|B],[3,6,C|D],[4,7,E|F]):-E\ is\ A+C,addlists(B,D,F).,\\ \qquad\langle 1:(4\ is\ 1+3),1:(7\ is\ 1+6)\rangle\}\end{array}\right)$$
$$\{addlists([X,X,R|S],[Y,Z,T|U],[4,7,V|W]):-V\ is\ R+T,addlists(S,U,W).,\ \langle 1:(4\ is\ X+Y),1:(7\ is\ X+Z)\rangle\}$$

Figure 4.7: *msg* of Versions `addlists_2`, `addlists_3`, `addlists_4` and `addlists_5`.

- **else** if for every evaluated external success $c \in T$ such that c is no longer sufficiently instantiated to be executed we can determine that its corresponding $c_1 \in \tau_1$ and $c_2 \in \tau_2$ are both *deterministic*,
  - then *residualize* all $c \in T$ being no longer sufficiently instantiated.
  - otherwise we cannot collapse A_1 and A_2.

Note that without such residualization, the code generated by the *msg* is not directly usable, since there are bindings in the original versions which are lost if we apply the code produced by the *msg*.

Example 4.6.1. *As we have already mentioned, all characteristic trees in Figure 4.5 are quasi-isomorphic. Therefore, they can be collapsed into one version. In Figure 4.7 we show the* msg *of both the code and the characteristic trees for versions* `addlists_2`*,* `addlists_3`*,* `addlists_4` *and* `addlists_5`*. In this figure, the scope of variables is local to each clause. Since* $\tau_2 \approx \tau_3 \approx \tau_4 \approx \tau_5$*, the* msg *does not produce any information loss. This can be easily verified by instantiating back*

the msg *with any of the call patterns. For instance, if we take* `addlists([X,X],[Y,Z],[4,7])` *and instantiate it with* `addlists([3,3|G],[1,4|H],I)` *we obtain the original clause (eighth clause of Figure 4.2). This fact can be easily verified by taking any pattern call and after instantiating with the* msg *the original clauses are retrieved. For example, if we take* `addlists_2`*, then*

$$\begin{aligned} C = \;& \{addlists([X,X],[Y,Z],[4,7]),\\ & \quad addlists([X,X,R|S],[Y,Z,T|U],[4,7,V|W]) : -V \ is \ R+T, addlists(S,U,W)\},\\ A_2 = \;& addlists([4,4|A],[0,3|B],[4,7|C]), \end{aligned}$$

and $instantiate(C, A_2)$ *is*

$$\begin{aligned} & \{addlists([4,4],[0,3],[4,7]),\\ & \quad addlists([4,4,R|S],[0,3,T|U],[4,7,V|W]) : -V \ is \ R+T, addlists(S,U,W)\}, \end{aligned}$$

i.e, $instantiate(C, A_2) \approx code(A_2)$.

Example 4.6.2. *Now, let us compute the* msg *of the generalized code and characteristic tree obtained in Example 4.6.1 with* `addlists_6`.

$$\frac{msg\left(\begin{array}{l}\{addlists([X,X],[Y,Z],[4,7]).,\langle 1:(4 \ is \ X+Y), 1:(7 \ is \ X+Z)\rangle\}\\ \{addlists([7,\ 1],[\ 1,5],[8,6]).,\langle 1:(8 \ is \ \ 7+\ 1), 1:(6 \ \ is \ \ 1+5)\rangle\}\end{array}\right)}{\{addlists([A,B],[C,D],[E,F]).,\langle 1:(E \ is \ A+C), 1:(F \ is \ B+D)\rangle\}}$$

$$\frac{msg\left(\begin{array}{r}\{addlists([X,X,R|S],[Y,Z,T|U],[4,7,V|W]) : -V \ is \ R+T, addlists(S,U,W).,\\ \langle 1:(4 \ is \ X+Y), 1:(7 \ is \ X+Z)\rangle\}\\ \{addlists([7,\ 1,R|S],[\ 1,5,T|U],[8,6,V|W]) : -V \ is \ R+T, addlists(S,U,W).,\\ \langle 1:(8 \ is \ \ 7+1), 1:(6 \ is \ \ 1+5)\rangle\}\end{array}\right)}{\begin{array}{r}\{addlists([A,B,G|H],[C,D,I|J],[E,F,K|L]) : -K \ is \ G+I, addlists(H,J,L).,\\ \langle 1:(E \ is \ A+C), 1:(F \ is \ B+D)\rangle\}\end{array}}$$

The msg *introduces some information loss*

$$\begin{aligned} C = \;& \{addlists([A,B],[C,D],[E,F]),\\ & \quad addlists([A,B,C|D],[E,F,G|H],[I,J,K|L]) : -K \ is \ G+C, addlists(D,H,L)\},\\ A_2 = \;& addlists([4,4|A],[0,3|B],[4,7|C]), \end{aligned}$$

and $instantiate(C, A_2)$ *is*

$\{addlists([4,4],[0,3],[4,7]),$
$\quad addlists([4,4,R|S],[0,3,T|U],[4,7,V|W]) :- V \ is \ R+T, addlists(S,U,W)\},$

i.e., instantiate$(C, A_2) \approx$ *code*(A_2).

Since `addlists_6` *is not (fully) isomorphic with the other versions, the* msg *introduces some information loss through the variables* E *and* F *in the new heads* `addlists([A,B],[C,D],[E,F])` *and* `addlists([A,B,G|H],[C,D,I|J],[E,F,K|L])`*. This information loss cannot be recovered by instantiate, since, for example, when instantiating the* msg `addlists([A,B],[C,D],[E,F])` *with the call pattern* `addlists([3,3|G],[1,4|H],I)` *we obtain* `addlists([3,3],[1,4],[E,F])`*, in which* E *and* F *are unbound variables. If we take the external successes which correspond to* `E is A+C` *and* `F is B+D` *we can verify that the original external successes were deterministic (indeed, all calls to* `is/2` *are deterministic). Thus, it is possible to collapse by residualization. As both external calls are no longer sufficiently instantiated, they are residualized.*

Residualized atoms are always placed before any other atom in the generalized clause, guaranteeing that after execution of such residual atoms at run-time, the clause as a whole is actually a variant of the original definition of the clause.

The resulting minimized program is shown in Figure 4.8. Residual atoms are underlined to distinguish them from the rest of atoms in body clauses.

4.7 Experimental Results

In this section we experimentally assess the impact of our proposed minimization. Most of the benchmarks considered contain calls to builtins which possibly generate bindings, such as `is/2`, and thus the existing partial evaluators which perform minimization [79, 80] would not be able to minimize them optimally.

In our experiments we use an unfolding strategy based on homeomorphic embedding (see, e.g., [71]) and which performs leftmost unfolding steps only. This guarantees the correctness of the partial evaluation process even in the presence of impure predicates. Note that the issue of redundant polyvariance may occur for any unfolding strategy. The global control used is based on homeomorphic

```
main(A,B,C,D,E,F,G,H,I,J,K,L,M,N,O) :- write(A),
   addlists_6([4,4|A],[0,3|B],[4,7|C]),
   addlists_6([3,3|D],[1,4|E],[4,7|F]),
   addlists_6([3,3|G],[1,4|H],I),
   addlists_6([1,1|J],[3,6|K],L),
   addlists_6([7,1|M],[1,5|N],O) .

addlists_1([],[],[]).
addlists_1([A|B],[C|D],[E|F]) :-
   E is A+C, addlists_1(B,D,F) .

addlists_6([A,B],[C,D],[E,F]) :-
   E is A+C, F is B+D.
addlists_6([A,B,G|H],[C,D,I|J],[E,F,K|L]) :-
   E is A+C, F is B+D,
   K is G+I, addlists_1(H,J,L) .
```

Figure 4.8: Specialization of `addlists/3` after Minimization with Residualization.

embedding and global trees [74]. Benchmarks have been run on an Intel Pentium 4, 3.4 GHz processor, with 512 Mb of RAM, and running a 2.6 Linux kernel.

4.7.1 The Benefits of Minimization

Table 4.1 shows the size reduction introduced by the minimization step after partial evaluation. Each benchmark program is evaluated using five different minimization criteria, as shown in the *Min Crit* column. Specialization history is used in *pure*, *nobinds*, and *bindings*, in order to consider two versions as locally equivalent, while *codemsg* directly applies the definition of structural equivalence for the same purpose. In particular, *pure* considers two versions as locally equivalent when their characteristic trees are identical. Of course, if external successes are included, these must be identical too. The criteria *nobinds* and *bindings* check for isomorphism of external successes instead. *Nobinds* only considers two exter-

Benchmark	Min Crit	Orig Preds	Minimization					
			Versions			Size (bytes)		
			PE	Min	Ratio	PE	Min	Ratio
datetime	pure	15	56/31	36/36	1.78	131377	102651	1.28
	nobinds			36/36	1.78		102836	1.28
	bindings			34/35	1.83		102331	1.28
	codemsg			34/35	1.83		102295	1.28
	residual			31/33	1.94		100976	1.30
flattrees	pure	2	33/16	22/22	1.50	226390	223320	1.01
	nobinds			22/22	1.50		223435	1.01
	bindings			22/22	1.50		223389	1.01
	codemsg			17/19	1.74		221513	1.02
	residual			16/18	1.83		220796	1.03
freeof	pure	3	93/8	35/35	2.66	292642	245262	1.19
	nobinds			35/35	2.66		245442	1.19
	bindings			32/35	2.66		245370	1.19
	codemsg			18/35	2.66		245334	1.19
	residual			8/35	2.66		245370	1.19
mmatrix	pure	3	70/11	18/34	2.06	58323	37061	1.57
	nobinds			18/34	2.06		37236	1.57
	bindings			18/34	2.06		37166	1.57
	codemsg			18/34	2.06		37131	1.57
	residual			11/30	2.33		31781	1.84
nrev	pure	2	41/3	3/3	13.67	25115	5261	4.77
	nobinds			3/3	13.67		5281	4.76
	bindings			3/3	13.67		5273	4.76
	codemsg			3/3	13.67		5269	4.77
	residual			3/3	13.67		5273	4.76
qsort	pure	3	168/50	68/68	2.47	232079	166288	1.40
	nobinds			50/50	3.36		131650	1.76
	bindings			50/50	3.36		131548	1.76
	codemsg			50/50	3.36		131497	1.76
	residual			50/50	3.36		131548	1.76
sublists	pure	4	29/19	27/27	1.11	101969	99986	1.02
	nobinds			27/27	1.11		100121	1.02
	bindings			19/19	1.58		95815	1.06
	codemsg			19/19	1.58		95795	1.06
	residual			19/19	1.58		95815	1.06
Overall				2.88 / 2.96			1.32 / 1.33	

Table 4.1: Minimization Ratios over Selected Benchmarks

nal successes c and c' as isomorphic when they generate no bindings, i.e., when $Allpos(c) = Allpos(c) = \emptyset$, while *bindings* applies the full power of Def. 4.4.5.

Finally, *residual* considers two versions as candidates for minimization when their characteristic trees are quasi-isomorphic, possibly residualizing calls to external predicates in the resulting program.

The number of predicates in the original program is shown in the column *Orig Preds*. The number of predicates in the specialized programs are shown under the column *Versions*. *PE* shows both the number of versions which are generated after partial evaluation (i.e., the effects of polyvariance) and the number of sets of predicates with quasi-isomorphic characteristic trees. The latter provides a lower bound on the number of predicates which the minimized program may have. *Min* shows the number of elements in the partition generated by the reunion phase of the minimization algorithm (local equivalence) and the number of elements in the partition after the splitting phase (global equivalence). Finally, *Ratio* shows the reduction ratio for each criteria compared to the number of versions produced by partial evaluation. The column *Size* compares the sizes of the compiled bytecode of programs minimized using the different criteria.

The last row, *Overall*, shows the weighted geometric mean (*wgm*) for ratios in terms of number of versions and size. Weights are number of versions and size of the *PE* column, respectively. In both cases, under the column *Min* we find the *wgm* of the *codemsg* criterion, which achieves the best results while still producing programs of maximal optimization. Under the column *Ratio* we find the *wgm* of the *residual* criterion, which achieves highest ratio.

As can be seen in the table, in most of the benchmarks considered, minimization is capable of considerably reducing the specialized program, both in terms of number of versions and of bytecode size. As it is to be expected, out of the four criteria which are guaranteed to produce programs of maximal optimization, i.e., *pure*, *bindings*, *nobinds*, and *codemsg*, the one which produces the best results is the latter. Among the three of them which take the minimization history into account—and which are more efficient in terms of specialization time—, the best is *bindings*, but it sometimes does not produce as good results as *codemsg*. The effects of the splitting phase are clear in many benchmarks, showing that, in effect, local equivalence does not imply global equivalence. This effect is notorious in the case of the *residual* criterion, since after the reunion phase the number of locally equivalent versions is equal to the number of sets of versions having quasi-isomorphic characteristic trees, however, in the splitting phase they

are split, producing a larger number of versions, as can be clearly seen in the case of the `freeof` and `mmatrix` benchmarks. Finally, for `datetime`, `flattrees` and `mmatrix`, *residual* is able to further reduce code size.

4.7.2 The Cost of Minimization

In Table 4.2 we can observe the cost, in terms of specialization time, introduced by the minimization, expressed in milliseconds. The (*Total*) time of the whole specialization process is shown, including the time of partial evaluation (*Analysys*), minimization (*Minim*) and code generation (*Code*) steps. A new minimization criteria is introduced, *nomin*, showing the time employed by partial evaluation without minimization. The *Slow* column shows the cost (slowdown) of performing this minimization post-processing.

Interestingly, the table shows that when minimization is employed, the code generation phase takes less time in most cases, since fewer versions need to be generated. This lowers the burden introduced by the minimization post-processing.

However, even in the worst case, the slowdown introduced is reasonable (1.85). As expected, using specialization history makes minimization faster than just applying the definition of structural equivalence. Given the fact that employing structural equivalence generates fewer versions than other criteria based on the specialization history, the *codemsg* criterion emerges as a very interesting one. Also, for the *residual* minimization criterion, the time spent in code generation is greater than for the rest of criteria, since it requires deciding which external successes need to be residualized.

4.7.3 Benefits of Minimization in Runtime

Table 4.3 shows how specialized programs behave in terms of runtime. Benchmark programs having residualized external predicates (for the *residual* minimization criterion) are marked with * in the table. Column *PE Time* shows the absolute run-time for the partially evaluated program. The rest of the columns show the speedup achieved for the minimized programs (for each different minimization criteria) w.r.t. *PE Time*. As can be seen in the table, in most benchmarks a small speedup is achieved (1.00 – 1.20), and no slowdown is produced in any case. As expected, in the case of programs with residualized external predicates,

Bench	Min Crit	Minimization Times (msec)				
		Total	**Analysis**	**Minim**	**Code**	**Slowdown**
datetime	nomin	556.52	475.33	0	81.19	1
	pure	632.90	486.13	61.19	85.59	1.14
	nobinds	634.30	476.13	72.79	85.39	1.14
	bindings	640.10	479.93	73.99	86.19	1.15
	codemsg	642.30	478.13	79.19	84.99	1.15
	residual	687.30	479.93	77.59	129.78	1.23
flattrees	nomin	299.55	232.56	0	66.99	1
	pure	395.14	230.97	107.78	56.39	1.32
	nobinds	396.34	231.57	108.58	56.19	1.32
	bindings	400.19	230.21	113.48	56.49	1.34
	codemsg	412.74	231.36	125.98	55.39	1.38
	residual	424.94	231.36	125.78	67.79	1.42
freeof	nomin	5732.93	5583.15	0	149.78	1
	pure	5833.11	5589.95	118.98	124.18	1.02
	nobinds	5844.11	5589.15	131.38	123.58	1.02
	bindings	5858.31	5573.35	160.38	124.58	1.02
	codemsg	5948.90	5595.15	230.97	122.78	1.04
	residual	6113.47	5613.95	221.97	277.56	1.07
mmatrix	nomin	316.15	271.76	0	44.39	1
	pure	356.55	272.76	48.39	35.39	1.13
	nobinds	367.14	274.56	57.39	35.19	1.16
	bindings	364.34	272.76	55.99	35.59	1.15
	codemsg	373.34	274.96	63.19	35.19	1.18
	residual	435.53	270.76	60.79	103.98	1.38
nrev	nomin	898.26	877.07	0	21.20	1
	pure	886.67	861.27	13.20	12.20	0.99
	nobinds	901.86	872.67	16.80	12.40	1.00
	bindings	898.86	870.27	16.40	12.20	1.00
	codemsg	903.86	874.67	17.20	12.00	1.01
	residual	916.26	873.87	17.20	25.20	1.02
qsort	nomin	9983.68	9745.12	0	238.56	1
	pure	10267.64	9778.91	282.96	205.77	1.03
	nobinds	10303.83	9768.12	337.75	197.97	1.03
	bindings	10339.03	9771.91	368.94	198.17	1.04
	codemsg	10401.82	9764.92	441.73	195.17	1.04
	residual	11241.69	9732.72	371.14	1137.83	1.13
sublists	nomin	401.94	293.56	0	108.38	1
	pure	647.70	295.35	278.56	73.79	1.61
	nobinds	651.90	297.75	281.36	72.79	1.62
	bindings	679.50	295.56	280.16	103.78	1.69
	codemsg	681.30	297.56	278.76	104.98	1.70
	residual	744.09	296.95	284.56	162.57	1.85

Table 4.2: Minimization Times for Selected Benchmarks

Benchmark	PE Time	Speedup				
		Pure	No Binds	Bindings	CodeMsg	Residual
datetime*	167.77	1.01	1.02	1.01	1.01	1.01
flattrees*	81.39	1.03	1.01	1.01	1.03	1.01
freeof	246.96	1.04	1.04	1.05	1.04	1.05
mmatrix*	1920.11	1.02	1.02	1.02	1.02	1.00
nrev	141.38	1.20	1.18	1.18	1.19	1.19
qsort	457.33	1.05	1.04	1.04	1.05	1.04
sublists	15501.44	1.00	1.00	1.00	1.00	1.00

Table 4.3: Speedup over Selected Benchmarks

the speedup achieved is usually smaller than for the other minimization criteria.

4.8 Discussion and Related Work

The problem of superfluous polyvariance has been tackled in the context of abstract multiple specialization in [130, 108], and in the context of partial evaluation of normal logic programs in [84]. This chapter presents a unifying view under which the minimization problems in both contexts are isomorphic.

The work in [84], reflected in the ECCE [79] partial evaluator, uses an internal table of *safe* builtins which basically correspond to instantiation and type tests and which are guaranteed (1) not to generate any bindings, and (2) to be deterministic. The minimization phase then would only allow collapsing two predicates in the same version if their characteristic trees are quasi-isomorphic and all the builtins executed are listed in the table of pure predicates.

The approach presented herein, and implemented in the `Ciao` system preprocesor, `CiaoPP` [54], can handle any external predicate, including non-safe builtins, and the notion of isomorphic external predicates can be satisfied for builtins which generate bindings and which are non-deterministic. Also, there is no need for a static table of builtins. Additionally, the technique automatically applies to any external predicates, for example other modules written by the user.

To the best of our knowledge, this work presents the first experimental evaluation of the benefits of post-minimization in partial evaluation. We have compared several criteria, with different cost and potential benefit. We have also applied

directly the definition of structural equivalence and discovered that it is also applicable in practice, in addition to the other criteria based on the specialization history. Finally, we have proposed a criteria which allows residualizing external calls. The experiments show that it is also applicable in practice and provides some further program reduction.

Part III

Poly-Controlled Partial Evaluation: Foundations

Chapter 5

Poly-Controlled Partial Evaluation

As mentioned in Chapter 3, the aim of partial evaluation (PE) is to specialize a program w.r.t. part of its input, which is known as the *static data*[85]. The quality of the code generated by partial evaluation greatly depends on the *control strategy* used. Traditional algorithms for partial evaluation of logic programs (LP) are parametric w.r.t. the *global control* and *local control* rules. The issue of devising good control rules has received considerable attention (see for example [71] and its references). However, the existence of sophisticated control rules which behave (almost) optimally for all programs is still far from reality. Furthermore, existing control rules focus on time-efficiency by trying to reduce the number of resolution steps which are performed in the residual program. Other factors, such as the size of and the memory required to run the residual program, are most often neglected, a relevant exception being the work in [34]. In addition to potentially generating larger programs, it is well known (see e.g. [121, 28]) that partial evaluation can slow-down programs due to lower level issues such as clause indexing, cache sizes, etc. Also, once a choice of global and local control rules is made[1], such a combination will be applied to all call patterns in the residual program. Obviously, in practice, it can be very useful to be able to use *different* specialization strategies for *different* call patterns, thus obtaining results that cannot be produced using traditional partial evaluation with any given

[1]From now on, we call a combination of a global and a local control rule a *specialization strategy*.

specialization strategy.

In this chapter we describe a framework for on-line partial evaluation which allows using different specialization strategies for different call patterns and can generate several candidate specializations. These specializations can then be empirically compared for efficiency, in terms of multiple factors such as size of the specialized program and time- and memory-efficiency of such specialized program.

The framework was first introduced in [110], and it is *self-tuning* in that, as mentioned above, it uses empirical evaluations for automatically selecting the best candidates by means of a *fitness function*. It is also *resource-aware* in that multiple factors, such as size of specialized programs and their memory consumption, can be taken into account by the fitness function in addition to the natural consideration of time-efficiency of the specialized programs. In [27], a self-tuning, resource aware *offline* specialization technique was introduced. The algorithm is based on mutation of annotations of offline partial evaluation. In contrast, our approach performs *online* partial evaluation, and thus it is fully automatic. To the best of our knowledge, there are no similar approaches for *online* PE.

5.1 The Dilemma of Controlling PE

As mentioned above, when specializing a program there exist many powerful specialization strategies to choose from. Unfortunately, there is no silver bullet, i.e., most control rules behave well with some programs, but no so well with others. Sometimes, choosing the wrong control rule can lead to obtaining a *slower* residual program or to a (considerably) *larger* residual program. But in other situations, the same control rules can achieve important speedups, or can lead to residual programs having the properties we are interested in.

For example, let us take the program from Listing 5.1. In this program, there is a call to the builtin `is/2`. Since the call `C is B+1` is not sufficiently instantiated to be executed (`B` is not yet bound to an arithmetic expression), it is required to use non-leftmost unfolding in order to jump over this call and unfold `q/1`. However, this unfolding generates the residual program shown in Listing 5.2. In this case, the residual code is less efficient than the original definition of `p/1`, since several calls to `is/2` may have to be speculatively performed until a success is found, if any.

Listing 5.1: Program `p/1` having an expensive call

```
p(B):- C is B+1, q(C).

q(1).
q(2).
q(3).
q(4).
q(5).
q(6).
```

Note that instead of a call to `is/2` we could be calling an external predicate performing an expensive computation.

Listing 5.2: Residual code of `p/1`

```
p(A) :- 1 is A + 1.
p(A) :- 2 is A + 1.
p(A) :- 3 is A + 1.
p(A) :- 4 is A + 1.
p(A) :- 5 is A + 1.
p(A) :- 6 is A + 1.
```

A similar example of generating a slower residual program is shown in Listing 5.3, borrowed from [80]. In this program, the predicate `inboth/3` takes three input arguments, the last two being lists, and checks whether the element passed as a first argument is a member of the two given lists.

Listing 5.3: The `inboth/3` example

```
member(X,[X|T]).
member(X,[Y|T]) :- member(X,T).

inboth(X,L1,L2) :- member(X,L1),
                   member(X,L2).
```

Let us partially evaluate this program w.r.t. the set of atoms {`inboth(a,L,[X,Y])`}. By using non-determinate non-leftmost unfolding, we obtain the residual program in Listing 5.4.

Listing 5.4: Residual code for `inboth/3`

```
member(a,[a|T]).
member(a,[Y|T]) :- member(a,T).

inboth(a,L,[a,Y]) :- member(a,L).
inboth(a,L,[X,a]) :- member(a,L).
```

If we execute both the original and the residual programs with the runtime query `inboth(a,[b,c,d,e,...,a],[X,Y])`, then we can see that the original program only executes once the expensive call to `member(a,[b,c,d,e,...,a])`, while the residual program does it twice.

The classical solution to these problems is to disable non-leftmost unfolding unless it is deterministic (SP [40, 42, 44], ECCE [84]), or to allow non-leftmost unfolding without left-propagation of bindings (PADDY [104], MIXTUS [115]). Some partial evaluators, for instance, SAGE [49, 48] do not prevent such work duplication. This can result in huge slowdowns (see, e.g., [14]).

Also, in the presence of *impure* predicates, non-leftmost unfolding can even produce incorrect results [2]. On the other hand, performing non-leftmost unfolding can provide important gains in other cases. See, for example, the program in Listing 5.5.

Listing 5.5: The `exponential/3` Example

```
exp(Base,Exp,Res):-
        exp_ac(Exp,Base,1,Res).

exp_ac(0,_,Res,Res).
exp_ac(Exp,Base,Tmp,Res):-
        Exp > 0,
        Exp1 is Exp - 1,
        NTmp is Tmp * Base,
        exp_ac(Exp1,Base,NTmp,Res).
```

If we specialize it w.r.t. the query `exp(Base,3,Res)`, enabling non-leftmost unfolding allows to unroll the recursive calls. The residual code, after some arithmetic simplifications[2], is shown in Listing 5.6.

[2]The specializer in `CiaoPP` actually performs such simplifications of arithmetic operations.

Listing 5.6: Residual Code of the `exponential/3` Example

```
exp(A,3,B) :-  B is A*A*A.
```

From these examples it is clear that the selected control rule directly affects the quality of the generated code. Also, it is not trivial to select the appropriate control rules, since, as we have seen, the same feature of a local control rule, i.e., whether to allow non-leftmost unfolding, can be beneficial for certain calls (atoms) and can be harmful for others.

Though one could argue that a good rule of thumb can be to only perform non-leftmost unfolding for determinate atoms, i.e., those which only unify with a single clause head, this heuristic does not guarantee to always achieve the best specialization possible: an atom whose resolution is not determinate can become deterministic later on, since maybe just one (or even none) of the derivations which contain such step is successful or incomplete (i.e., all the rest are failing derivations). For example, in the program of Listing 5.5, although the unfolding is deterministic, this could be easily converted into non-deterministic by changing the first clause of `exp_ac/4` to `exp_ac(Exp,_,R,R):- Exp = 0.` Note that the problem of deciding whether an atom is deterministic is undecidable: it can always happen that an SLD tree which contains several non-failing derivations at some depth, contains at most one non-failing derivation in the next depth level.

Another related problem when performing partial evaluation is known as *loss of indexing*. In order to be more efficient, most Prolog systems index clauses according to their first argument [35], i.e., if the first argument of the current goal is instantiated, the clause head matching this goal can quickly be found. If this is not the case, then all clauses have to be checked one by one looking for a matching clause head. This is analogous to indexing in database systems and can provide an important performance boost when searching over a large set of clauses. For instance, let us take the program in Listing 5.7, borrowed from [27]. In this example, we have a collection of facts represented by `p/2`, where indexing is performed over its first argument, and as long as the first argument in the call to `p/2` is instantiated we will benefit from the speedups of indexing.

If we specialize this program, then we can obtain the program in Listing 5.8, where, as we can see, indexing over the first argument has been lost, and as a consequence, this program will perform slower than the original one.

Listing 5.7: Example using clause indexing

```
index_test(f(_),Y,Z) :- p(Y,Z).

p(a,1).
p(b,2).
p(c,3).
p(d,4).
p(e,5).
p(f,6).
p(g,7).
p(h,8).
```

Listing 5.8: Specialiation of `index_test/3`. Clause indexing has been lost

```
index_test(f(_), a, 1).
index_test(f(_), b, 2).
index_test(f(_), c, 3).
index_test(f(_), d, 4).
index_test(f(_), e, 5).
index_test(f(_), f, 6).
index_test(f(_), g, 7).
index_test(f(_), h, 8).
```

Another related pitfall of partial evaluation is the explosion of code that can be generated in the residual program, as we have already seen in Chapter 4. This explosion of code is unacceptable if disk space or memory are important factors, and can even harmful in terms of speed, due to effects such as cache miss [34].

Many more pitfalls of partial evaluation can be found in [121], most of which are still valid today.

5.1.1 A Motivating Example

We now show in Listing 5.9 a program which defines the predicate `main/3` containing calls to the predicates `exp/3` and `p/1` defined before:

Listing 5.9: A Motivating Example

```
main(A,B,C):-
        exp(B,2,C),
        p(A).
```

In Listing 5.10 we can see the residual code obtained when specializing this program w.r.t. the query `main(A,B,C)` using leftmost unfolding. Note that none of the calls to the builtin predicate `is/2` are sufficiently instantiated to be executed at specialization time. Since only leftmost unfolding is allowed, the unfolding trees computed are not very deep, resulting in a large number of residual predicates.

Listing 5.10: Result with Leftmost Unfolding

```
main(A,B,C) :- D is 1*B,
        exp_ac_1(1,B,D,C),
        p_1(A).

exp_ac_1(1,A,B,C) :- D is B*A,
        exp_ac_2(0,A,D,C).

exp_ac_2(0,_1,A,A).

p_1(A) :- B is A+1, q_1(B).

q_1(1).
q_1(2).
q_1(3).
q_1(4).
q_1(5).
q_1(6).
```

On the other hand, if we choose to enable non-leftmost unfolding, we obtain the residual program shown in Listing 5.11, where only an SLD tree has been required, and thus no auxiliary predicates are defined.

Unfortunately, neither the program in Listing 5.10 nor the one in Listing 5.11 is optimal. This is because, in order to achieve an optimal result, non-leftmost un-

folding should be used for atoms for predicate `exp/3`, but only leftmost unfolding should be used for atoms for predicate `p/1`.

Listing 5.11: Result with Non-leftmost Unfolding

```
main(A,B,C) :- D is 1*B, C is D*B, 1 is A+1.
main(A,B,C) :- D is 1*B, C is D*B, 2 is A+1.
main(A,B,C) :- D is 1*B, C is D*B, 3 is A+1.
main(A,B,C) :- D is 1*B, C is D*B, 4 is A+1.
main(A,B,C) :- D is 1*B, C is D*B, 5 is A+1.
main(A,B,C) :- D is 1*B, C is D*B, 6 is A+1.
```

Note that although the rule of thumb discussed above for non-leftmost unfolding happens to provide good results in this example, clearly there is no unfolding strategy which uniformly obtains the optimal results in all cases.

As we have seen, choosing the right specialization strategy is a tough task. In this thesis, we propose a *poly-controlled* partial evaluation (*PCPE*) approach, in which several specialization strategies and heuristics can coexist, leaving to the framework the decision of which strategy is the most appropriate w.r.t. the users needs (speed, size of the residual program, etc.).

5.2 Poly-Controlled Partial Evaluation

Poly-controlled partial evaluation (*PCPE*) takes as input a program P, a set $\mathcal{A}$ of atoms describing the initial call patterns, and a set $\mathcal{CS}$ of specialization strategies. As output, PCPE can generate potentially multiple specialized programs. The PCPE process starts from an initial *configuration*, and repeatedly transforms it into a *child* configuration until a *final configuration* is reached. Since we allow the existence of multiple *sepcialization strategies*, a non-final configuration can have several *children* configurations. Depending on the approach used, a different number of configurations will be expanded and thus, different specialized programs will be obtained. These concepts are formalized below.

Definition 5.2.1 (configuration)**.** *A* configuration *is a pair $\langle S, H\rangle$, where S is a set of atoms and H is a set of tuples of the form $\langle A, A', \langle G, U\rangle\rangle$.*

The set S contains the atoms to be specialized and H contains the specialization history: for each previously specialized atom A we store, in addition to A

itself, the result A' of applying an abstraction operator G to it, and the specialization strategy $\langle G, U\rangle$ which has been applied on A'.

Correctness of the algorithm requires that each A' is an *abstraction* of A, i.e., $A = A'\theta$. The atom A is stored for precise predicate renaming, while U is stored in order to use exactly such unfolding rule during code generation (see Def. 5.2.8). Finally, G will be needed later by some pruning techniques (see Chapter 8).

Definition 5.2.2 (initial, intermediate and final configurations)**.** *A configuration is* initial *when it is of the form $\langle \mathcal{A}, \emptyset\rangle$. A configuration is* final *when it is of the form $\langle \emptyset, H\rangle$. Configurations that are not final are called* intermediate *configurations.*

As customary in PE, we consider the existence of an arbitrary function, which we call $TakeOne$, that given an intermediate configuration $\langle S, H\rangle$, decides the atom A to be specialized at each configuration among those in S, denoted $A = TakeOne(S)$.

We assume the existence of a function *atoms* that extracts the generalized atoms out of the tuples in H.

Definition 5.2.3 (atoms)**.** *Let $\langle S, H\rangle$ be a configuration s.t.* $H = \{\langle A_1, A'_1, \langle G_{1i_1}, U_{1j_1}\rangle\rangle, \ldots, \langle A_n, A'_n, \langle G_{1i_n}, U_{nj_n}\rangle\rangle\}$*. Then the set of* atoms *of H is defined as $atoms(H) = \{A'_1, \ldots, A'_n\}$.*

In an abuse of notation, when referring to abstraction operators we simply write $A' = G(A, H)$ instead of $A' = G(A, atoms(H))$. Finally, given a configuration $T = \langle S, H\rangle$, we use $\tau = CS(T)$ to denote that $A = TakeOne(S)$, $A' = G(A, H)$ and $\tau = U(P, A')$.

Definition 5.2.4 (PCPE-step)**.** *Let $T = \langle S, H\rangle$ be an intermediate configuration, and let $A = TakeOne(S)$. Let $CS{=}\langle G, U\rangle$ be a specialization strategy. Then a* PCPE-step *for T using CS generates a new configuration $T' = \langle S', H'\rangle$, denoted $T \rightsquigarrow_{CS} T'$, s.t.*

- $S' = (S - \{A\}) \cup \{B \in leaves(CS(T)) \mid \forall\ \langle C, \_, \_\rangle \in H\ .\ B \not\approx C\}$
- $H' = H \cup \{\langle A, A', \langle G, U\rangle\rangle\}$*, with* $A' = G(A, H)$

where the function leaves *collects the atoms in the bodies of $resultants(CS(T))$.*

Definition 5.2.5 (resultants)**.** *Let P be a program, let A be an atom, let U be an unfolding rule s.t. $U(P, A)$=τ.*

Then $resultants(\tau) = \{A\theta_1 \leftarrow R_1, \ldots, A\theta_n \leftarrow R_n\}$ where $\leftarrow R_1, \ldots, \leftarrow R_n$ are goals chosen from all non-failing leaves of τ, and θ_i is the substitution associated with the derivation from $\leftarrow A$ to $\leftarrow R_i$.

Given $T \rightsquigarrow_{CS} T'$, we say that T' is a *child* of T. PCPE-steps are organized into PCPE-paths.

Definition 5.2.6 (PCPE-path)**.** *A* PCPE-path *consists of a sequence $T_0 : T_1 : \ldots : T_p$ of configurations and a sequence $CS_1 : CS_2 : \ldots : CS_p$ of specialization strategies s.t. for $i = 1..p$, $T_i \rightsquigarrow_{CS_{i+1}} T_{i+1}$.*

We say that a PCPE-path $T_0 \rightsquigarrow_{CS_1} \ldots \rightsquigarrow_{CS_p} T_p$ is *complete* iff T_0 is an initial configuration and T_p is a final configuration. A configuration T' is *reachable* from a configuration T iff there is a path of the form $T \rightsquigarrow_{CS_1} \ldots \rightsquigarrow_{CS_p} T'$, $p \geq 0$.

PCPE-paths can be organized into *PCPE-trees*.

Definition 5.2.7 (PCPE-tree)**.** *A* PCPE-tree *is a tree where each node corresponds to a configuration, and which satisfies:*

- *The root node is an initial configuration.*
- *Leaves are final configurations.*
- *There is an arc from node T to node T' iff there is a specialization strategy $CS \in \mathcal{CS}$ s.t. $T \rightsquigarrow_{CS} T'$.*

From a final configuration we can obtain a *PCPE specialized program*. As usual in partial evaluation, during code generation we will rename apart atoms in order to avoid the independence requirement [44]. We use *rename* to refer to a procedure which assigns a fresh predicate name to each atom $A'_i \in H$ and performs appropriate renamings (using the pairs of atoms A_i, A'_i in the tuples of H) in the head and body of residual rules so that each program point uses a correct (and as optimized as possible) version.

Definition 5.2.8 (PCPE specialized program, solution, SP)**.** *Let $T = \langle \emptyset, H \rangle$ be a final configuration. Then H is called a* solution *of PCPE. Also, the* PCPE specialized program *P_T obtained from T, denoted $P_T = SP(T)$, is*

$P_T = \bigcup_{\langle A_i, A'_i, \langle G_i, U_i \rangle \rangle \in H} rename(resultants(U_i(P, A'_i)), H)$

As already mentioned, PCPE can generate several specialized programs. In fact, from any intermediate configuration we can reach a set of final configurations, each one corresponding to a possibly different specialized program.

Definition 5.2.9 (solutions)**.** *Let T be a configuration. The set of* solutions *for T is defined as $solutions(T) = \{SP(T') \mid T'$ is reachable from $T \wedge T'$ is final$\}$.*

Depending on the particular implementation of the PCPE algorithm, we could generate more than one specialized program. In order to choose the best specialized program, we can apply an *evaluation* step which uses a *fitness function Fit* to assess how good each specialized program P_T is is w.r.t. the original program P. The fitness function returns a value in $[\,0 \ldots \infty)$, with larger fitness values indicating better programs. Also, values smaller than one indicate that the specialized program is worse than the original one.

Definition 5.2.10 (maximal fitness value, mfv)**.** *Let T be an intermediate configuration. Let Fit be a fitness function. Then the* maximal fitness value *of T w.r.t. Fit, denoted $mfv_{Fit}(T)$, is defined as $max(\{Fit(P_{T_1}), \ldots, Fit(P_{T_p})\})$, where $solutions(T) = \{P_{T_1}, \ldots, P_{T_p}\}$ and $max(R)$ returns the largest value in the R set.*

We can now define a PCPE-path leading to a solution of maximal fitness.

Definition 5.2.11 (PCPE-path of maximal fitness)**.** *A complete PCPE path $T_0 \leadsto_{CS_1} \ldots \leadsto_{CS_p} T_p$ is of* maximal fitness *w.r.t. a fitness function Fit iff for $i = 0..p$, $mfv_{Fit}(T_i) = mfv_{Fit}(T_0)$.*

Note that for all pairs of configurations T and T', if T' is reachable from T then $mfv_{Fit}(T) \geq mfv_{Fit}(T')$, for any fitness function Fit. In a path of maximal fitness, we always perform PCPE-steps which preserve the maximal fitness value. A specialized program P' obtained by PCPE is of *maximal fitness* if $Fit(P') = mfv_{Fit}(T_0)$.

In the following two sections we consider two possible implementations of PCPE. The first algorithm is called PCPE_{one}, is greedy and it obtains only one specialized program. However, we cannot guarantee that the obtained program is of maximal fitness. The second one, called PCPE_{all}, traverses the complete PCPE-tree, and then evaluates all obtained specialized programs in order to select a program of maximal fitness.

5.3 A Greedy PCPE Algorithm

Algorithm 2 One-Solution Poly-Controlled Partial Evaluation Algorithm (PCPE_{one})

Input: Program P
Input: A finite set of atoms S
Input: A finite set of specialization strategies $\mathcal{CS}$
Input: Selection function $Pick$
Output: A partial evaluation for P and S, encoded by H_n

1: $i = 0$
2: $H_0 = \emptyset$
3: $S_0 = S$
4: **repeat**
5: $\quad A_i = TakeOne(S_i)$
6: $\quad \langle G,U \rangle = Pick(A_i, H_i, \mathcal{CS})$
7: $\quad A'_i = G(H_i, A_i)$
8: $\quad \tau_i = U(P, A'_i)$
9: $\quad H_{i+1} = H_i \cup \{\langle A_i, A'_i, \langle G, U \rangle\rangle\}$
10: $\quad S_{i+1} = (S_i - \{A_i\}) \cup \{A \in leaves(\tau_i) \mid \forall\ \langle B, \_, \_\rangle \in H_{i+1}\ .\ B \not\approx A\}$
11: $\quad i = i + 1$
12: **until** $S_i = \emptyset$

Algorithm 2, also referred to as PCPE_{one}, shows a *one-solution poly-controlled* partial evaluation (PCPE) algorithm. We refer to this algorithm as poly-controlled because it allows the use of multiple specialization strategies, possibly applying different strategies for different atoms. We also refer to it as one-solution since it is greedy, traversing only one complete PCPE-path, thus obtaining only one solution. We will see in Section 5.4 a *search-based* PCPE algorithm that obtains several candidate solutions.

One difference between this algorithm and the greedy PE algorithm seen in Chapter 3 is that, rather than receiving as input an abstraction operator and an unfolding rule, it receives *a set* $\mathcal{CS}$ of specialization strategies. The choice of the specialization strategy to apply during the handling of each atom is performed

by the *Pick* function, which given an atom A, a specialization history H, and $\mathcal{CS}$, picks up a specialization strategy $\langle G,U\rangle \in \mathcal{CS}$.

The algorithm produces only one *final configuration* $\langle \emptyset, H_n\rangle$. The output of the algorithm is H_n, i.e., $H_n = \mathsf{PCPE}_{one}\ (P, S, \mathcal{CS}, Pick)$.

We define a function *peel* in order to be able to compare the configurations of Algorithm 1 (PE) and Algorithm 2 (PCPE_{one}).

Definition 5.3.1 (peel)**.** *Let* $H = \{\langle A_1, A'_1, \langle G_{1i}, U_{1j}\rangle\rangle, \ldots, \langle A_k, A'_k, \langle G_{ki'}, U_{kj'}\rangle\rangle\}$. *Then* $peel(H) = \{\langle A_1, A'_1\rangle, \ldots, \langle A_k, A'_k\rangle\}$.

Lemma 5.3.2. *Let P be a program, let S be a finite set of atoms, let G be a local control rule and let U be a global control rule.*

Then $\mathsf{PE}(P, S, G, U) = peel(\mathsf{PCPE}_{one}\ (P, S, \{\langle G, U\rangle\}, Pick))$.

Proof. As noted before, the differences between Algorithm PE and Algorithm PCPE_{one} are:

1. Algorithm 2 introduces a new function *Pick* to select a specialization strategy to be applied to a given atom. However, in our case this function is deterministic and will always select $\langle G, U\rangle$.

2. Tuples in the solution obtained by PCPE_{one} are different from those in the solution obtained by PE. However, by using the peel function we get rid of the extra information.

□

Clearly, different choices for the *Pick* function will result in different specialized programs. It is important to note that the finer-grained control of poly-controlled partial evaluation can potentially produce specialized programs which are hard or even impossible to obtain by using off-the-shelf specialization strategies. Also, the addition of the *Pick* function conceptually makes the poly-controlled partial evaluation algorithm being composed of three levels of control, the local control, the global control, and the *search control*, which is determined by the function *Pick*. Note that the inclusion of the history as an input argument to *Pick* allows to make hopefully more informed decisions.

Depending on the particular choices of control strategies made by the *Pick* function, the solution obtained by PCPE_{one} could, in fact, be obtained by using

always the same specialization strategy, or using different specialization strategies for different atoms. In our context, solutions of the first kind are called *pure*, while the rest of solutions are called *hybrid*. Pure solutions can be obtained by traditional partial evaluation, hybrid solutions cannot. We extend this notion of purity to configurations.

Definition 5.3.3 (pure and hybrid configurations)**.** *Let $T = \langle S, H\rangle$ be a configuration. Then T is* pure *iff* $\forall\langle A_i, A'_i, CS_i\rangle \in H,\ \forall\langle A_j, A'_j, CS_j\rangle \in H\ .\ CS_i = CS_j$.
A configuration that is not pure is called a hybrid *configuration.*

Lemma 5.3.4. *Let P be a program and let S be a finite set of atoms. Let $\mathcal{CS}$ be a set of specialization strategies. Let $T = \langle\emptyset, H\rangle$ be a* pure final *configuration s.t $H = \mathsf{PCPE}_{one}\ (P, S,\ \mathcal{CS}, Pick)$. Then $\exists\langle G, U\rangle \in \mathcal{CS}$ s.t. $\mathsf{PE}\ (P, S, G, U) = peel(H)$.*

Proof. Since $T = \langle\emptyset, H\rangle$ is a *pure* configuration, all tuples in H are of the form $\langle A_i, A'_i, \langle G, U\rangle\rangle$ for a given specialization strategy $\langle G, U\rangle \in \mathcal{CS}$. Thus *Pick* is a deterministic function that always returns a tuple $\langle G, U\rangle$. By Lemma 5.3.2, it follows then that $PE(P, S, G, U) = peel(H)$. □

5.4 A Search-based PCPE Algorithm

PCPE_{one} can provide better specializations than those achievable by traditional partial evaluation algorithms by assigning different specialization strategies to different atoms. However, the improvements achieved rely on the behavior of the function *Pick*. Unfortunately, choosing a good *Pick* function can be a very hard task. Another alternative is, instead of deciding *a priori* the specialization strategy to apply to each atom, to generate several (or even all) candidate partial evaluations and then decide *a posteriori* which specialized program to use. In the extreme, this can be done by traversing the complete PCPE-tree.

Algorithm 3 shows a search-based algorithm (PCPE_{all}) that generates a *set* of final configurations $\{\langle\emptyset, H_1\rangle, \ldots, \langle\emptyset, H_n\rangle\}$. In other words,

$\mathsf{PCPE}_{all}\ (P, S, \mathcal{CS}) = \{H_1, \ldots, H_n\}$.

Obviously, in general we will be interested in selecting only one specialized program out of all final programs obtained. Clearly, generating all possible can-

Algorithm 3 All-solutions Search-based Partial Evaluation Algorithm (PCPE_{all})

Input: Program P

Input: A finite set of atoms S

Input: A finite set of specialization strategies $\mathcal{CS}$

Output: A finite set of partial evaluations *Sols*

1: $i = 0$
2: $H_0 = \emptyset$
3: $S_0 = S$
4: $\mathsf{create}(Confs)$; $Confs = \mathsf{push}(\langle S_0, H_0 \rangle,\ Confs)$
5: $Sols = \emptyset$
6: **repeat**
7: $\quad \langle S_i, H_i \rangle = \mathsf{pop}(Confs)$
8: $\quad A_i = TakeOne(S_i)$
9: $\quad Candidates = \mathcal{CS}$
10: $\quad$ **repeat**
11: $\quad\quad Candidates = Candidates - \{\langle G, U \rangle\}$
12: $\quad\quad A'_i = G(H_i, A_i)$
13: $\quad\quad \tau_i = U(P, A'_i)$
14: $\quad\quad H_{i+1} = H_i \cup \{\langle A_i, A'_i, \langle G, U \rangle\rangle\}$
15: $\quad\quad S_{i+1} = (S_i - \{A_i\}) \cup \{A \in leaves(\tau_i) \mid \forall\ \langle B, \_, \_ \rangle \in H_{i+1}\ .\ B \not\approx A\}$
16: $\quad\quad$ **if** S_{i+1}=$\emptyset$ **then**
17: $\quad\quad\quad Sols = Sols \cup \{H_{i+1}\}$
18: $\quad\quad$ **else**
19: $\quad\quad\quad \mathsf{push}(\langle S_{i+1}, H_{i+1} \rangle, Confs)$
20: $\quad\quad$ **end if**
21: $\quad$ **until** $Candidates = \emptyset$
22: $\quad i = i + 1$
23: **until** $\mathsf{empty\_stack}(Confs)$

didate specialized programs is more costly than computing just one. However, selecting the best candidate a posteriori allows to make much more informed decisions than selecting it a priori, as in Algorithm 2. Another difference with Algorithm 2 is that Algorithm 3 employs two additional data structures:

- *Confs*, which contains the configurations which are currently being explored.
- *Sols*, which stores the set of solutions found by the algorithm.

As it is well known, the use of different data structures for *Confs* provides different traversals of the PCPE-tree. The actual implementation uses both a stack (depth-first traversal) and a queue (breadth-first traversal). As we will see later, sometimes we will want to traverse the PCPE-tree in either way. Note that Algorithm 3 does not work with single configurations but rather with stacks (queues) of configurations. The process terminates when the stack (queue) of configurations to handle is empty, i.e. all final configurations have been reached.

Lemma 5.4.1. *Let P be a program and let S be a finite set of atoms. Let $\mathcal{CS}$ be a set of specialization strategies. Then, for* any *arbitrary Pick function,*
$\mathsf{PCPE}_{one}\ (P, S, \mathcal{CS}, Pick) \in \mathsf{PCPE}_{all}\ (P, S, \mathcal{CS})$.

Proof. In each step of the algorithm PCPE_{one}, for a given configuration $T = \langle S, H\rangle$, *Pick* will select a given tuple $CS_{ij} \in \mathcal{CS}$, and apply it to a selected atom $A \in S$. PCPE_{all}, on the other hand, will apply *all* control strategies in $\mathcal{CS}$ for all configurations. Therefore, CS_{ij} must be one of the control strategies PCPE_{all} will use with T. □

Lemma 5.4.2. *Let P be a program and let S be a finite set of atoms. Let $\mathcal{CS}$ be a set of specialization strategies. Then $\forall H_k \in \mathsf{PCPE}_{all}\ (P, S, \mathcal{CS}) \exists$ a* Pick *function s.t. $H_k = \mathsf{PCPE}_{one}\ (P, S, \mathcal{CS}, Pick)$.*

Proof. Let $H_k = \{\langle A_1, A'_1, \langle G_{1i}, U_{1j}\rangle\rangle, \ldots, \langle A_k, A'_k, \langle G_{ki'}, U_{kj'}\rangle\rangle\}$.

Then *Pick* is defined as follows:

$$Pick(A_p, H_p, \mathcal{CS}) = \begin{cases} \langle G_{1i}, U_{1j}\rangle & \text{if } p = 1 \\ \ldots & \\ \langle G_{ki'}, U_{kj'}\rangle & \text{if } p = k \end{cases}$$

□

Corollary 5.4.3. *Let P be a program and let S be a finite set of atoms. Let $\mathcal{CS}$ be a set of specialization strategies. Then $\forall H_k \in \mathsf{PCPE}_{all}\ (P, S, \mathcal{CS}) \Leftrightarrow \exists$ a* Pick *function s.t. $H_k = \mathsf{PCPE}_{one}\ (P, S, \mathcal{CS}, Pick)$.*

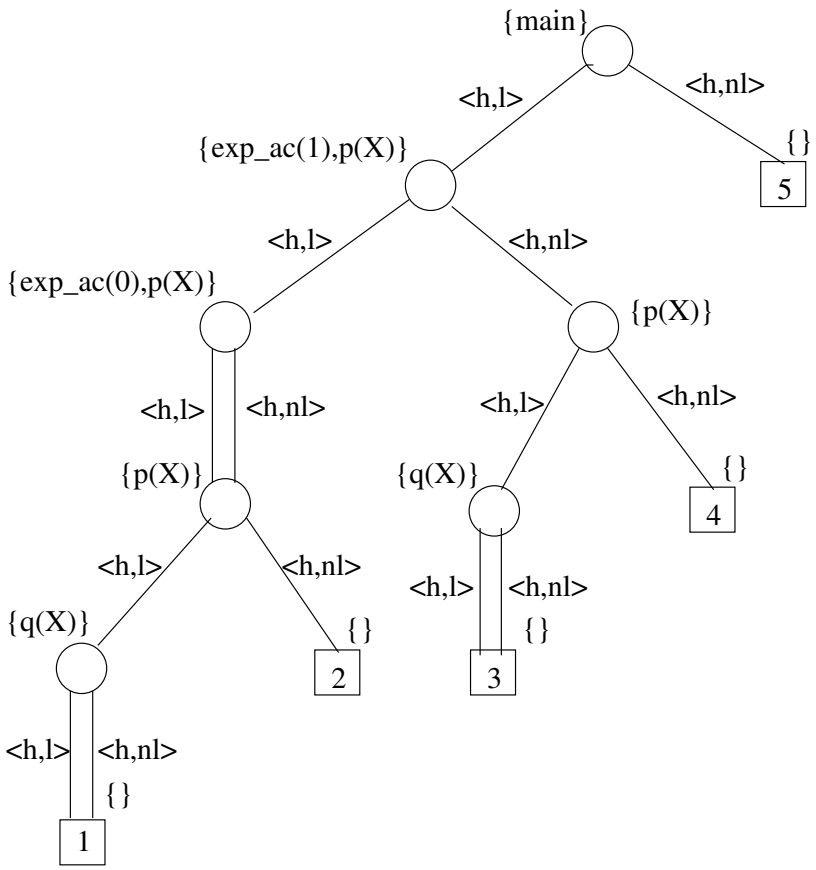

Figure 5.1: Complete PCPE-tree for the Motivating Example 5.9

Proof. It follows immediately from Lemma 5.4.1 and Lemma 5.4.2. □

Lemma 5.4.4. *Let P be a program and let S be a finite set of atoms. Let $\mathcal{CS}$ be a set of specialization strategies. Then* $\forall\langle G, U\rangle \in \mathcal{CS}$. PE *(P,S,G,U)* $\in$ *peel*(PCPE_{all} $(P, S, \mathcal{CS})$)

Proof. By Lemma 5.3.4, PE (P, S, U, G) = *peel*(PCPE_{one} $(P, S, \mathcal{CS}, Pick)$) for a deterministic *Pick* function that always returns $\langle G, U\rangle \in \mathcal{CS}$. Then, by Lemma 5.4.1, PCPE_{one} $(P, S, \mathcal{CS}, Pick) \in \mathsf{PCPE}_{all}$ $(P, S, \mathcal{CS})$. □

5.5 Searching for All Specializations

Consider again the example in Listing 5.9. Consider also two unfolding rules, one performing leftmost unfolding only, and the other one performing also non-

leftmost unfolding, and one abstraction operator *hom_emb* based on homeomorphic embedding (see Chapter 3), s.t.

$\mathcal{CS} = \{\langle hom\_emb, leftmost\rangle, \langle hom\_emb, nonleftmost\rangle\}$.

By applying PCPE_{all} we get five different specialized programs. In particular, *Solution1* corresponds to the program in Listing 5.10 and *Solution5* to the program in Listing 5.11. In addition, our algorithm also produces three other candidate programs which are hybrid in the sense that they use different control rules for different atoms, and thus cannot be achieved using $\langle hom\_emb, leftmost\rangle$ nor $\langle hom\_emb, nonleftmost\rangle$ only.

The PCPE-tree for this example is shown in Figure 5.1. There, intermediate configurations are represented by a circle, while final configurations are represented by a square. As can be seen, the whole search space for the example consists of 12 configurations, 7 of which are intermediate and 5 are final. The latter ones correspond to different candidate solutions, as already mentioned.

Each configuration is adorned with the set S of atoms yet to be handled. Each node can have two children, one per specialization strategy in $\mathcal{CS}$, which are indicated with arcs. Arcs are labeled either `<h,l>`, for $\langle hom\_emb, leftmost\rangle$ or `<h,nl>` for $\langle hom\_emb, nonleftmost\rangle$. The set of nodes already handled is not shown explicitly in each node, but it is implicitly represented by traversing the tree from each node upwards up to the root, since an atom is handled in each node. For example, in the case of *Solution3*, the history is $\{\langle q(B), q(B), \langle h, nl\rangle\rangle$, $\langle p(A), p(A), \langle h, l\rangle\rangle, \langle exp\_ac(1, A, B, C), exp\_ac(1, A, B, C), \langle h, nl\rangle\rangle, \langle main(A, B, C),$ $main(A, B, C), \langle h, l\rangle\rangle\}$. Also, some nodes only have one descendant linked by two arcs to its parent. This indicates that the two specialization strategies considered produce equivalent configurations, reducing the search space.

Table 5.1 provides a comparison of the different candidate solutions together with the original program. The first column indicates the program we refer to in each row. The second column provides an indication of the run-time efficiency of the different programs. This time has been obtained by running a million times the query `main(8,9,Result)` and subtracting the time required by an empty loop which performs a million iterations. The third column compares the sizes of the different programs. This size is in number of bytes of the program compiled into bytecode using Ciao-1.13 and after subtracting the size of an empty program. Finally, the last two columns compare the run-time and code-size of the different

Program	Runtime	Size	Speedup	Code Reduc
Original	5890	1606	1.00	1.00
Solution1	3652	1596	1.61	1.01
Solution2	5138	1543	1.15	1.04
Solution3	2931	1379	2.01	1.16
Solution4	3962	1326	1.49	1.21
Solution5	7223	1321	0.82	1.22

Table 5.1: Comparison of Solutions

programs with that of the original program.

As it can be seen, not all programs obtained by partial evaluation are necessarily faster than the original one. In particular, *Solution5*, the one obtained using non-leftmost unfolding for all cases is less efficient than the original one. This is indicated by an speedup lower than 1, which is 0.82 in this case. On the other hand, the speedup obtained by *Solution1* is 1.61, but it is still far from the fastest program, which is *Solution3* with an speedup of 2.01. As regards code size, in this particular case all solutions achieved are smaller than the original program, though, as seen in Chapter 4, in some cases partial evaluation can produce programs which are significantly larger than the original one. The smallest program is *Solution5*, with a code reduction of 1.22, but which happens to be the slowest program of all, including the original one.

If both the speedup and code reduction factors are taken into account, the most promising programs are probably *Solution3* and *Solution4*, neither of which are achievable by using one unfolding rule for all atoms. If code size is not a very pressing issue, then *Solution3* is probably the best one, but otherwise *Solution4* should be used, since a relative small increase in program size provides significant time performance improvement. The choice between the two solutions mentioned will depend on the fitness function used, which can put more emphasis in one factor or another.

5.6 Self-Tuning, Resource-Aware PE

Though Algorithm PCPE_{all} can be used to automatically generate a large number of candidate specialized programs to choose from, we need some mechanism to automatically select just one of them since, obviously, the goal of partial evaluation is to obtain a specialized program, not many. There are certainly several criteria which can be used in order to decide how good a specialized program is. The framework we propose in this work is *resource-aware* since it can take the following criteria into account.

Time efficiency: currently we are measuring speedup w.r.t. the original program. In this case, we need a set of test cases which are representative of the class of run-time queries which will be performed. Another possibility to be explored is the use of static cost analysis. Cost analysis can aim at obtaining upper or lower bounds on computational cost or even average cost.

Size of compiled code: fairly easy to measure. It can be an important factor if the program will run on devices with limited resources, as is the case in embedded systems and pervasive computing. Also, even in cases where code size is not much of an issue, it can happen that different specialized programs have similar time-efficiency but some of them can be significantly larger than others.

Memory-consumption: it can be of interest when resources are scarce, similarly to the case of size of compiled code.

Our framework is fully automatic, i.e., there is no need for human intervention in order to decide which is the best among the candidate specializations. We refer to this as a *self-tuning* approach. A *fitness function* assigns a numeric value in $[0 \ldots \infty)$ to each candidate specialization, reflecting how good the corresponding program is w.r.t. the original program. Larger fitness values indicating better programs. Also, values smaller than one indicate that the specialized program is worse than the original one.

The framework is parametric w.r.t. the fitness function so that the method can be applied with different aims in mind. Sometimes we may be interested

in achieving code which is as time-efficient as possible, whereas in other cases space-efficiency can be a primary aim. It is important to note that this search-based approach to partial evaluation is also of interest when only run-time is taken into account. Even in such case there is no specialization strategy alone which is guaranteed to always produce the most-efficient code for all compilers and architectures.

The fitness functions implemented in our framework that take into account the above resource-aware criteria are fully described in Appendix A.

5.7 Correctness of PCPE

The original definition of partial evaluation in Chapter 3 (borrowed from [85]) does not mention an unfolding rule. However, as observed by [44], given an atom A and a program P, there may be infinitely many different partial evaluations of A in P. An unfolding rule is then used to obtain only one of them. We reformulate the definition of partial evaluation using an unfolding rule below.

Definition 5.7.1 (partial evaluation)**.** *Let P be a definite program and let A be an atom. Let U be an unfolding rule. Then the* partial evaluation *of A in P using U is defined as $resultants(U(P, A))$.*

If $\mathcal{A}$ is a set of atoms, then a partial evaluation *of $\mathcal{A}$ in P using U is defined as $\bigcup_{A \in \mathcal{A}} resultants(U(P, A))$.*

Our goal is to prove that PCPE_{all} is a *correct* algorithm for partial evaluation of a program P w.r.t. some goal Q. The central result proved by Lloyd and Shepherdson in [85] on the correctness of partial evaluation is Theorem 3.2.6 from Chapter 3. This theorem implies that, given a program P, an unfolding rule U and some goal Q, a *correct* algorithm for partial evaluation of P w.r.t. Q must compute an *independent* set of atoms $\mathcal{A}$ s.t. if P' is a partial evaluation of $\mathcal{A}$ in P using U then $P' \cup \{Q\}$ must be $\mathcal{A}$-closed.

Gallagher [44] observes that it is possible to drop the condition that a partial evaluation algorithm returns an *independent* set of atoms $\mathcal{A}$ by using a *renaming transformation* ρ, which renames every atom in $\mathcal{A}$ by giving it a *fresh* predicate symbol and keeping its arguments unmodified. The set of atoms to be specialized thus become independent without introducing any precision loss. Given a program P and a set of atoms $\mathcal{A}$ s.t. P' is a partial evaluation of $\mathcal{A}$ in P, a *renaming*

for P', denoted $\rho(P')$, is obtained by applying a renaming transformation to all atoms in $\mathcal{A}$ and mapping atoms inside the bodies of the residual program clauses of P' (during code generation) to the correct version of the renamed predicate.

As observed in [44], the set $\mathcal{A}' = \rho(\mathcal{A})$ is independent, and thus, when checking the correctness of a partial evaluation algorithm, we only need to check that $R' \cup \{Q\}$ is $\mathcal{A}'$-closed, for some goal Q.

In the definitions and lemmas below, we consider the algorithm PCPE_{all} for poly-controlled partial evaluation which takes a definite program P, a finite set of atoms S (initial queries), a non-empty finite set $\mathcal{CS} = \{\langle G_1, U_1\rangle, \ldots, \langle G_n, U_m\rangle\}$, producing non-empty set of solutions $\{H_1, \ldots, H_k\}$, where each solution H_i is a set of tuples of the form $\langle A, A', \langle G, U\rangle\rangle$.

The following definitions and lemmas help in proving the correctness of PCPE_{all}.

As we noted before, in traditional partial evaluation there is an implicit unfolding rule U. Thus, given a set of atoms $\mathcal{A}$ and a program P, we can generate a partial evaluation of $\mathcal{A}$ in P using U. However, since PCPE can handle *several* unfolding rules, we need to know which unfolding rule is to be used with each atom when generating a partial evaluation. The following function extracts sets of tuples containing such information out of a solution.

Definition 5.7.2 (annotated atoms)**.**
Let H_k be a solution s.t. $H_k = \{\langle A_1, A'_1, \langle G_{1i_1}, U_{1j_1}\rangle\rangle, \ldots, \langle A_n, A'_n, \langle G_{ni_n}, U_{nj_n}\rangle\rangle\}$. Then the set of annotated atoms *of H_k, denoted by $annotated\_atoms(H_K)$, is defined as $annotated\_atoms(H_K) = \{\langle A'_1, U_{1j_1}\rangle, \ldots, \langle A'_n, U_{nj_n}\rangle\}$.*

Now, we can define a poly-controlled partial evaluation in terms of annotated atoms.

Definition 5.7.3 (poly-controlled partial evaluation)**.** *Let P be a definite program and let $\langle A, U\rangle$ be an annotated atom. Then the* poly-controlled partial evaluation *of $\langle A, U\rangle$ in P is defined as $resultants(U(P, A))$.*

If $\mathcal{H}$ is a finite set of annotated atoms, then a poly-controlled partial evaluation *of $\mathcal{H}$ in P is $\bigcup_{\langle A,U\rangle\in\mathcal{H}} resultants(U(P, A))$.*

As can be seen, in poly-controlled partial evaluation each annotated atom is unfolded using its respective unfolding rule. We now formalize a procedure for synthesizing an unfolding rule from a set of annotated atoms.

Definition 5.7.4 (combine)**.**

Let P be a definite program. Let $\mathcal{H} = \{\langle A_1, U_1\rangle, \ldots, \langle A_n, U_n\rangle\}$ be a set of annotated atoms. Then combine($\mathcal{H}$) builds an unfolding rule $\hat{U}_{\mathcal{H}}$ s.t.

$$\hat{U}_{\mathcal{H}}(P, A) = \begin{cases} U_1(P, A) & \text{if } A \approx A_1 \\ \ldots & \\ U_n(P, A) & \text{if } A \approx A_n \end{cases}$$

Note that $\hat{U}_{\mathcal{H}}$ is a *partial* function built *a posteriori*, by considering the set of annotated atoms of a given solution. This rule is instrumental in proving the correctness of PCPE. Note also that there exists one specific unfolding rule $\hat{U}_{\mathcal{H}}$ per PCPE solution.

We now prove that the residual program obtained by poly-controlled partial evaluation could be obtained by traditional partial evaluation using $\hat{U}_{\mathcal{H}}$ as an unfolding rule.

Lemma 5.7.5. *Let P be a definite program and let H_k be a solution. Let $\mathcal{H}_{H_k} = annotated\_atoms(H_k)$. Let $\hat{U}_{\mathcal{H}} = combine(\mathcal{H}_{H_k})$. Let $\mathcal{A} = atoms(H_k)$. Let P' be the partial evaluation of $\mathcal{A}$ in P using $\hat{U}_{\mathcal{H}}$. Let P'' the poly-controlled partial evaluation of $\mathcal{H}_{H_k}$ in P.*

Then $P' = P''$.

Proof. Trivially holds by using the function combine as a *glue*. In other words,

- By definition 5.7.1, $P' = \bigcup_{A_{ki} \in \mathcal{A}} resultants(\hat{U}_{\mathcal{H}}(P, A_{ki}))$.
- By definition 5.7.3, $P'' = \bigcup_{\langle A_{ki}, U_{ki}\rangle \in \mathcal{H}_{H_k}} resultants(U_{ki}(P, A_{ki}))$.

Thus $P' = P''$ since $\forall \langle A_{ki}, U_{ki}\rangle \in \mathcal{H}_{H_k}.\hat{U}_{\mathcal{H}}(P, A_{ki}) = U_{ki}(P, A_{ki})$ by definition 5.7.4.

□

Lemma 5.7.6. *Let P be a definite program and let H_k be a solution. Let $\mathcal{H}_{H_k} = annotated\_atoms(H_k)$. Let $\mathcal{A} = atoms(H_k)$. Let P'' be the poly-controlled partial evaluation of $\mathcal{H}_{H_k}$ in P.*

Then P'' is $\mathcal{A}$-closed.

Proof. First note that H_k is a PCPE solution, i.e., it is obtained from a final configuration $\langle S_k, H_k \rangle$ s.t. $S_k = \emptyset$.

By Lemma 5.7.5, we can build an synthetic unfolding rule $\hat{U}_{\mathcal{H}} = combine(\mathcal{H}_{H_k})$, and obtain a partial evaluation P' of $\mathcal{A}$ in P using $\hat{U}_{\mathcal{H}}$ s.t. $P' = P''$, and prove by contradiction that P' is $\mathcal{A}$-closed.

Let us assume that P' is not $\mathcal{A}$-closed. Then, $\exists A_{ki} \in \mathcal{A}$ s.t. $resultants(\hat{U}_{\mathcal{H}}(P, A_{ki}))$ is not $\mathcal{A}$-closed. In other words, $pred(A_{ki}) \in \mathcal{A} \wedge A_{ki}$ is *not* an instance of an atom in $\mathcal{A}$. By the constructive nature of PCPE_{all}, A_{ki} must then belong to S_k. This is a contradiction since, as we said above, $S_k = \emptyset$. □

Theorem 5.7.7. *Let P be a definite program and let H_k be a solution. Let $\mathcal{H}_{H_k} = annotated\_atoms(H_k)$. Let $\mathcal{A} = atoms(H_k)$ and let $\mathcal{A}' = \rho(\mathcal{A})$. Let P' be a poly-controlled partial evaluation of $\mathcal{H}_{H_k}$ in P and let $R' = \rho(P')$.*

Then for all goals Q such that $R' \cup \{Q\}$ is $\mathcal{A}'$-closed

- *$P \cup \{Q\}$ has a SLD-refutation with computed answer θ iff $R' \cup \{Q\}$ has a SLD-refutation with computed answer θ.*
- *$P \cup \{Q\}$ has a finitely-failed SLD-tree iff $R' \cup \{Q\}$ has a finitely-failed SLD-tree.*

Proof. It follows immediately from Lemma 5.7.6, and from Lloyd's Theorem 3.2.6 (see [85]) recalled in Chapter 3. □

5.8 Some Notes on the Termination of PCPE

One question remaining to be answered is whether PCPE is guaranteed to terminate. In other words, given a set $\mathcal{CS}$ of specialization strategies, where for each $\langle G, U \rangle \in \mathcal{CS}$ we know that both G and U guarantee termination, does the combination of different local control and global control rules put at risk the termination of the whole algorithm?

Let us consider Algorithm 3. This algorithm has two nested loops, and some calls to external procedures. Let us analyze first the calls to external procedures.

- We assume that simple procedures (create, pop, push) dealing with data structures terminate.

- The calls to G and U in lines 12 and 13 correspond to the application of the abstraction and unfolding functions, respectively. It is our initial assumption that these procedures guarantee termination on their own.

Now we can analyze the termination of each of the loops.

- In order to guarantee the termination of the inner loop (line 21), the set $Candidates$ must be finite. As can be seen in line 9, this set contains all specialization strategies in $\mathcal{CS}$, and $\mathcal{CS}$ is a finite set.

- In order to guarantee the termination of the outer loop (line 23), we have to guarantee that the stack $Confs$ becomes empty at some point in the execution of the algorithm. Note that, besides the initial assignment of $Confs$ (line 4), configurations are pushed into $Confs$ in line 19. This occurs every time a new configuration $\langle S_i, H_i \rangle$ is generated s.t. $S_i \neq \emptyset$. In other words, we have to ensure that for all possible configurations, at some point $S_i = \emptyset$. An atom A is introduced in S_i (line 15) if A is not a variant of a (previously visited) atom B s.t. $\langle B, \_, \_ \rangle \in H_i$. Note that atoms to be added to S are obtained from the leaves of the SLD-tree resulting from the unfolding of a *generalization* of the currently selected atom. Since all global control rules are terminating, this means that they can generalize a (possibly infinite) set of atoms into a finite one. I.e., at some point all atoms resulting from unfolding a generalization of a selected atom will be already in the set of visited atoms, and they will not be added to S, so S will become empty at some point in time.

  In other words, let us assume that there is a combination of local and global control rules s.t. S never becomes empty, i.e., there is an infinite PCPE-path $T_0 \leadsto_{\langle G_{1i_1}, U_{1i_1} \rangle} T_1 \leadsto_{\langle G_{2i_2}, U_{2i_2} \rangle} \ldots$. If we extract the infinite sequence $G_{1i_1} : G_{2i_2} : \ldots$ of global control rules applied from T_o, then it is possible to find at least one infinite subsequence containing the same control rule G_i. But this contradicts the initial assumption that global control rules guarantee termination on their own. Thus, there cannot be such an infinite subsequence, i.e., there cannot be an infinite PCPE-path, so S must become empty at some point in time.

ID	Abstraction	Unfolding
c1	*hom_emb*	*one_step*
c2	*hom_emb*	*df_hom_emb_as*
c3	*dynamic*	*one_step*
c4	*dynamic*	*df_hom_emb_as*

Table 5.2: Specialization Strategies

5.9 Preliminary Evaluation

In order to perform a preliminary assessment of the benefits and practicality of search based poly-controlled partial evaluation, we have conducted a series of experiments using the `CiaoPP` [106, 55] system.

Although the search-based approach presented in Section 5.4 above is definitely appealing, it is worth investigating whether it can actually produce better specializations than traditional partial evaluation (PE) and also whether it produces too large a number of candidate specialized programs, even for small input programs. In our evaluation we have compared two extreme cases: PE vs PCPE_{all}.

In our experiments, we have used a set $\mathcal{CS} = \{c1, c2, c3, c4\}$ of specialization strategies shown in Table 5.2, where the *hom_emb* abstraction operator is based on homeomorphic embedding [71, 81] and flags atoms as potentially dangerous (and are thus generalized) when they homeomorphically embed any of the previously visited atoms. Then, *dynamic* is the most abstract possible global control rule, which abstracts away the value of all arguments of the atom and replaces them with distinct variables. As explained in Chapter 3, the unfolding rule *one_step* is the simplest possible unfolding strategy which always performs just one unfolding step for any atom. Finally, *df_hom_emb_as* is an unfolding strategy based on homeomorphic embedding. We have chosen these particular specialization strategies since, on the one hand, they guarantee termination, and, on the other hand, they allow us to contrast aggressive and conservative unfolding. In this way, we expect to obtain more heterogeneous candidate solutions.

When testing PCPE_{all} for the specialization strategies mentioned above, we have found out that the approach copes with many of the benchmarks by Lam & Kusalik [69]. However, these benchmarks are of relatively little interest to our technique, since many of them can be fully unfolded. Thus, in general,

Benchmark	Compiled size	#versions
example_pcpe	5504	27
permute	4687	70
nrev	4623	117
qsortapp	5390	40
sublists	5638	58
relative	5909	61

Table 5.3: Size and Number of Versions of Benchmarks

traditional partial evaluation obtains good results, and the solutions provided by PCPE are *pure* solutions, achievable by PE. However, in practice, it is often the case that programs being partially evaluated cannot be fully unfolded since the static information available is not sufficient to do so. Table 5.3 shows the size in bytes of the compiled bytecode of each benchmark, as well as the number of candidate solutions being generated by the PCPE approach. Further details on these benchmarks can be found in Appendix B. In order to keep the number of candidate solutions reasonable, in most cases we have provided specialization queries containing very few static data. As a result, in some of the programs the speed-up achieved by partially evaluating the program is not very high using any of the strategies, since little information is known at specialization time. The specialization queries used in our experiments for each benchmark are shown in Table 5.4.

As we have mentioned previously, the solutions computed by PCPE are evaluated using a fitness function, and the solution of maximal fitness is considered to be the output of the whole algorithm. In our experiments, we have used the fitness functions Speedup, Bytecode and Balance:

Speedup compares programs based on their time-efficiency, measuring run-time speedup w.r.t. the original program.

Bytecode compares programs based on their space-efficiency, measuring reduction of size of compiled bytecode w.r.t. the original program.

Balance is a combination of the Speedup and Bytecode fitness functions. It tries to achieve small and fast programs.

Benchmark	Specialization query
example_pcpe	main(A,B,2,D)
nrev	rev([_,_|L],R)
permute	permute([1,2,3,4,5,6],L)
qsortapp	qsort([_,_|L],R)
sublists	sublists(A,B,C)
relative	relative(john,X)

Table 5.4: Specialization Queries Used in our Experiment

Further details on these fitness functions can be found in Appendix A.

5.9.1 Benefits of PCPE

We now try to evaluate whether PCPE can actually produce better results than traditional PE. Tables 5.5, 5.6, and 5.7 show how PCPE solutions behave when compared to the solutions obtained by traditional PE, using different fitness functions. In order to be as informative as possible, the best solution obtained by PCPE has been compared against all specialized programs obtained by PE when running *every* specialization strategy.

Each table shows the benchmark being considered, the fitness value obtained by the solution of poly-controlled partial evaluation, and its *composition* (columns *c1* through *c4*, see below), and the fitness value of every solution found by traditional partial evaluation using the different specialization strategies. Note that all fitness functions are defined in such a way that the original program has fitness 1, and values greater than one indicate improvements over the original program, whereas values less than one indicate that the considered program is worse than the original program (under the corresponding criterion). In the case of the PCPE solution, columns *c1* through *c4* describe the percentage of atoms in the selected best solution whose specialization behaviour can be achieved using the corresponding specialization strategy. Note that the addition of the values of *c1* through *c4* for a given program will be 100 or more. The latter can occur because different controls can result in exactly the same specialization for certain atoms. A value of a 100 in a given column means that such best solution can be obtained by traditional partial evaluation by using the corresponding specialization

Benchmark	Best PCPE					Fitness Trad PE			
	c1	**c2**	**c3**	**c4**	**Fit**	**c1**	**c2**	**c3**	**c4**
example_pcpe	75	50	50	25	1.96	0.91	0.57	1.01	0.49
permute	0	100	0	0	5.26	0.75	5.14	1.01	2.00
nrev	57	57	0	14	1.20	0.51	0.77	0.99	0.91
qsortapp	50	50	83	67	1.06	0.86	0.87	0.99	0.94
sublists	57	43	71	43	1.08	0.97	0.99	0.98	0.88
relative	0	0	0	100	14.08	0.98	14.05	0.98	14.02

Table 5.5: Preliminary Results of PCPE (SPEEDUP).

strategy.

Table 5.5 shows the results achieved when we use SPEEDUP as a fitness function.

- In general, speedup values in most cases should be greater than 1. However, since we are providing very little static information to the partial evaluation algorithms, in the case of *nrev*, *qsortapp*, and *sublist* the speedup achieved w.r.t. the original program is very small, and in many cases (especially in traditional partial evaluation) the specialized program is somewhat slower than the original one.

- Speedups are however evident in the *relative* and *permute* benchmarks, since they can be fully unfolded. In these two cases, and considering only SPEEDUP as the fitness function, the solution obtained by PCPE is a solution that can be obtained by traditional PE. In the case of *permute*, it is achieved by *c2*, i.e., using *hom_emb* as a global control rule and *df_hom_emb_as* as a local control rule. This is indicated by the 100 in *c2* column. We can also observe that the speedup of both the PCPE solution and the solution obtained by traditional PE using such control rules are pretty much the same and the difference lies only in timing errors during the experiments, since they correspond to the same program. In the case of *relative*, PCPE obtains two (best) solutions, one containing a 100 in column *c2*, not shown in this table, and one containing a 100 in column *c4*. As can be seen in the table, this speedup value is very similar to the one obtained

Benchmark	Best PCPE					Fitness Trad PE			
	c1	**c2**	**c3**	**c4**	**Fit**	**c1**	**c2**	**c3**	**c4**
example_pcpe	75	50	50	25	1.22	0.82	1.15	0.98	0.39
permute	25	50	50	50	1.15	0.37	0.00	0.98	0.80
nrev	20	60	60	80	0.98	0.55	0.29	0.98	0.76
qsortapp	33	67	67	83	0.98	0.78	0.43	0.98	0.66
sublists	100	25	100	25	0.98	0.98	0.52	0.98	0.61
relative	20	60	40	60	1.17	0.66	0.89	0.98	0.13

Table 5.6: Preliminary Results of PCPE (Bytecode).

by traditional PE using such control rules since, again, they correspond to the same code.

- For this particular fitness function, the rest of benchmarks are the interesting ones, since the solution obtained by PCPE cannot be obtained by PE, as there is no 100 in any column. In all cases the PCPE solution gets a better fitness value than any of the solutions provided by traditional PE, i.e., the obtained specialized program is faster. The PCPE solution in these cases is between 6% and 95% faster than the corresponding best PE solution. Note that this result is interesting in itself: PCPE can achieve better results than any single control rule even in the case where only speedup is taken into account.

These experiments were performed on a 1.5 GHz PowerPC G4 processor, with 1Gb of RAM, running on a Darwin 8.5 kernel. Times are given in milliseconds and are computed as the arithmetic mean of five runs.

Table 5.6 compares PCPE and traditional PE using Bytecode as a fitness function.

- As can be expected from the selected set of benchmarks, the solutions obtained by PE have a fitness value below 1 in most cases, indicating that the specialized programs are larger than the original one. This usually is due to the fact that these benchmarks contains just a few predicates, and partial evaluation creates many new specialized predicates which then cannot be unfolded very much.

Benchmark	Best PCPE					Fitness Trad PE			
	c1	**c2**	**c3**	**c4**	**Fit**	**c1**	**c2**	**c3**	**c4**
example_pcpe	75	50	50	25	1.54	0.87	0.81	0.99	0.44
permute	40	40	40	40	1.30	0.54	0.14	1.01	1.29
nrev	60	60	0	40	1.12	0.52	0.48	0.98	0.82
qsortapp	50	50	83	67	1.00	0.81	0.61	0.99	0.78
sublists	100	25	100	25	1.01	1.00	0.70	0.99	0.73
relative	20	60	40	80	4.05	0.80	3.55	0.98	1.33

Table 5.7: Preliminary Results of PCPE (BALANCE).

- When programs can be fully unfolded, as is the case of *relative* and *permute*, the use of *df_hom_emb_as* as a local control rule usually achieves such full unfolding. In the case of *permute*, the fitness is almost 0 for *c2* since the final fully unfolded program is much larger than the original one.

- Furthermore, programs produced using *c3*, i.e., *dynamic* as a global control rule and *one_step* as a local control rule, are indeed isomorphic to the original program. In this case, fitness values are slightly lower than 1 (0.98) due to predicate renamings, which creates slightly larger predicate names.

- As can be seen in the table, most of the programs obtained by PCPE are not achievable using PE. The only exception is *sublists*, where the best PCPE solution corresponds to the original program. Thus, it seems that PCPE is able to find a solution that is smaller than any of the solutions found by PE.

Finally, Table 5.7 shows the results achieved by using BALANCE as a fitness function.

- As can be seen, most of PCPE solutions cannot be obtained via traditional PE, with the exception of the solution for *sublists*, where the solution of maximal fitness coincides with not partially evaluating the program, i.e., the original program.

- In most cases, PCPE obtains better fitness values than any of the solutions

Benchmark	Specialization Time						
	PE			PCPE			PCPE
	Spec	Code	Total	Spec	Code	Total	/PE
example_pcpe	26	43	69	111	304	415	6
permute	1153	744	1897	1271	1242	2513	1
nrev	16	27	44	453	1166	1619	37
qsortapp	22	39	61	153	425	578	10
sublists	22	41	63	206	649	854	14
relative	216	166	382	1038	1187	2225	6

Table 5.8: Cost of PCPE (Specialization Time in msecs.)

obtained by PE, meaning that PCPE outperforms PE in most cases when both time- and space-efficiency are simultaneously considered.

5.9.2 Cost of PCPE

We now evaluate the cost of performing PCPE when compared to PE. Though one can argue that, in the case of compile-time specialization, the time required to specialize a program is not very important, the results presented here provide some information of the additional compile-time cost of PCPE when compared to PE. Depending on the situation, the developer may choose to spend more resources on specializing the program in return for a (hopefully) better specialized program.

Specialization in both traditional and poly-controlled partial evaluation involves a phase commonly referred to as *analysis* (corresponding to algorithms PCPE_{one} and PCPE_{all} described in this chapter), and another phase for code generation. Table 5.8 shows the times (expressed in milliseconds) spent for both approaches during these two phases, under columns `Spec` and `Code`, respectively. The last column shows the ratio between PCPE and PE.

- As can be seen, the burden introduced by the PCPE approach is usually acceptable.
- In many cases this overhead is directly related with the amount of candidate solutions generated by the algorithm. Thus, *nrev* is the benchmark

Benchmark	Evaluation Time(Speedup)			
	PE	PCPE		
		Spec	Eval	Total
example_pcpe	69	415	11540	11955
permute	1897	2513	19971	22484
nrev	44	1619	66692	68312
qsortapp	61	578	17159	17737
sublists	63	854	77104	77958
relative	382	2225	17007	19232

Table 5.9: Total Cost of PCPE (Speedup) (Time in msecs.)

where the ratio *PCPE/PE* is bigger, since there are 117 candidate solutions produced by the algorithm.

- Observe that in those cases that can be fully unfolded, i.e., *permute* and *relative*, specialization time is usually high in both approaches, so the ratio *PCPE/PE* is not very high.

- In the rest of benchmarks this ratio ranges between 6 and 14.

However, PCPE involves an additional step of evaluation after code generation which is not required in PE. In this step, all candidate solutions are evaluated using the corresponding fitness function, and the solution of maximal fitness represents the output of the algorithm for the given input queries. Note that there may exist several solutions of maximal fitness.

When evaluating all candidates, the fitness function used for such purpose makes an important difference in the time required by such phase. In the case of Bytecode, we need to compile each candidate solution and compare the sizes of the compiled code of all of them. Even though this involves disk accesses, the comparison among solutions can be done pretty quickly, and thus, the increment in time due to evaluation is acceptable, as shown in Table 5.10. In most cases, evaluation takes between two and four times the time spent in the previous two phases.

However, when the fitness function involves measuring time-efficiency, i.e. in Speedup and Balance, we need to run each specialized program a number of

Benchmark	Evaluation Time(Bytecode)			
	PE	PCPE		
		Spec	Eval	Total
example_pcpe	69	415	1230	1645
permute	1897	2513	3587	6100
nrev	44	1619	5826	7444
qsortapp	61	578	2332	2909
sublists	63	854	4016	4870
relative	382	2225	5173	7399

Table 5.10: Total Cost of PCPE (Bytecode) (Time in msecs.)

iterations in order to obtain more accurate measurements, thus increasing the time spent in evaluation (see tables 5.9 and 5.11). In our implementation, we have a constant K for estimating the desired amount of time we want to evaluate each candidate. By running the original program for K milliseconds, we estimate the number of iterations to be run for each of the final candidates. By increasing or decreasing this constant K, we increase or decrease the time spent by the evaluation step of our algorithm. In this way, we have a trade-off between the time spent in this phase, and the accuracy of the obtained solution. For our experiments, we set this constant to 500 milliseconds. As a result, we spend roughly about 500 milliseconds evaluating each candidate solution.

5.10 Highlights of PCPE

In this chapter we have introduced a framework for on-line partial evaluation which allows using different specialization strategies for different atoms, obtaining results that are not achievable by traditional partial evaluation. The framework is self-tuning, employing resource-aware *fitness functions* to select the best solutions from a resulting set of candidate solutions.

Among the main advantages of PCPE we can mention:

It can obtain better solutions than traditional PE: as we have seen in Section 5.9, our preliminary experiments have shown that PCPE can produce *hybrid* solutions whose fitness value is better than any of the solutions

achievable by traditional PE, for a number of different resource-aware fitness functions.

It is a resource-aware approach: in traditional PE, existing control rules focus on time-efficiency by trying to reduce the number of *resolution steps* which are performed in the residual program. Other factors such as the size of the compiled specialized program, and the memory required to run the residual program are most often neglected—some relevant exceptions being the works in [34],[27]—. In addition to potentially generating larger programs, it is well known that partial evaluation can slow-down programs due to lower level issues such as clause indexing, cache sizes, etc. PCPE, on the other hand, makes use of *resource aware* fitness functions to choose the best solution from a set of candidate solutions.

It is not yet another control strategy: the topic of control strategies for partial evaluation has received considerable attention. As already mentioned, finding an optimal control strategy is not trivial, since it always seems possible to find a counterexample for any heuristic. However, it is important to note that PCPE is not a control strategy, but a new framework allowing the co-existence and cooperation of any set of control strategies. In fact, PCPE will benefit from any further research on control strategies.

It is more user-friendly: existing partial evaluators usually provide several global and local control rules, as well as many other parameters (global trees, computation rules, etc.) directly affecting the quality of the obtained solution. For a novice user, it is extremely hard to find the right combination of parameters in order to achieve the desired results (reduction of size of compiled code, reduction of execution time, etc.). Even for an experienced user, it is rather difficult to predict the behavior of partial evaluation, especially in terms of space-efficiency (size of the residual program). PCPE allows the user to simultaneously experiment with different combinations of parameters in order to achieve a specialized program with the desired characteristics.

It performs online partial evaluation: as opposed to other approaches (e.g. [27]), PCPE performs *online* partial evaluation, and thus it is fully auto-

Benchmark	Evaluation Time(Balance)			
	PE	PCPE		
		Spec	Eval	Total
example_pcpe	69	415	12887	13302
permute	1897	2513	24408	26920
nrev	44	1619	73538	75157
qsortapp	61	578	19886	20463
sublists	63	854	82898	83752
relative	382	2225	22755	24980

Table 5.11: Total Cost of PCPE (Balance) (Time in msecs.)

matic and can use a more powerful set of specialization strategies.

Unfortunately, PCPE is not the panacea, and it has a number of disadvantages. The main drawback of this approach is that, when implemented as a search-based algorithm, its search space suffers from an inherent exponential blowup since given a configuration, the number of children configurations that can be derived from it can be as high as the number of specialization strategies in the set $\mathcal{CS}$ considered. Also, the specialization time of PCPE is higher than its PE counterpart. In the third part of this thesis we deal with these problems.

5.11 Related Work

As regards related work, the work in [27] is probably the most related one. There, a self-tuning, resource aware *off-line* specialization technique is introduced. The algorithm is based on mutation of annotations for offline partial evaluation. In contrast, our approach performs *on-line* partial evaluation, and thus can take advantage of the great body of work available for *on-line* partial evaluation of logic programs. To the best of our knowledge, there are no similar approaches for *on-line* partial evaluation.

Chapter 6

Heterogeneity of Solutions

In Chapter 5 we have introduced poly-controlled partial evaluation (PCPE). As we saw, PCPE is a flexible approach for specializing logic programs. Its main characteristic is that it can use *different* specialization strategies for different call patterns.

After getting acquainted for the first time with the basic idea of poly-controlled partial evaluation, probably two questions come up immediately to our mind:

1. does PCPE provides a wide range of solutions? I.e., since PCPE starts from an initial configuration T_0 and produces a set $\{H_1, \ldots, H_n\}$ of candidate solutions, is this set *heterogeneous* enough to offer us a wide set of candidate solutions to choose from?

2. is PCPE feasible in practice? I.e., since there is an exponential blowup of the search space[1], is it possible to perform some pruning in order to deal with realistic programs without losing the interesting solutions?

In this chapter we address the first question, providing experimental results to help us justify our allegations. The third part of this thesis is devoted to deal with the second question.

[1]This happens when PCPE is implemented using the search-based algorithm PCPE_{all}.

6.1 Choosing an Adequate Set of Specialization Strategies

The question of whether the solutions obtained by PCPE are heterogeneous w.r.t. their fitness values depends, in a great deal, on the particular choice of specialization strategies to be used, as well as on the arity of the set $\mathcal{CS}$ of specialization strategies. We can expect that by choosing control rules different enough, the candidate solutions will be also very different, and viceversa. For example, let us say we only use *det* and *lookahead* as unfolding rules, where both *det* and *lookahead* are purely determinate [44, 42], i.e., they select atoms matching a single clause head, the difference being that *lookahead* uses a "look-ahead" of a finite number of computation steps to detect further cases of determinacy [44]. Given that both unfolding rules are based on determinate unfolding, and this is considered a very conservative technique, it is highly probable that this particular choice of unfolding rules will not contribute to find heterogeneous solutions. This will we empirically shown in Chapter 7.

A better idea will be then to choose one unfolding rule that is conservative, and another one that is aggressive. An example of an aggressive local control rule would be one performing *non-leftmost* unfolding.

The same reasoning can be done when selecting the global control rules, we could select one rule that is very precise—while guaranteeing termination—, and a very imprecise global control rule.

Note that our framework is very flexible in this sense: as we will see in Chapter 11, in the graphical interface of `CiaoPP` we remove the user of the burden of selecting adequate control strategies. Instead, the user can select among different levels of aggressiveness for specialization strategies. However, more experienced users can still play with all control strategies available in `CiaoPP` from an expert interface.

Benchmark	Input Query
example_pcpe	main(_,_,2,_)
permute	permute([1,2,3,4,5,6],L)
nrev	rev([_,_,_,_|L],R)
advisor	what_to_do_today(_,_,_)
relative	relative(john,X)
ssuply	ssupply(_,_,_)
transpose	transpose([[_,_,_,_,_,_,_,_,_],_,_],_)

Table 6.1: Input Queries Used to Specialize Each Benchmark

6.2 Heterogeneity of the Fitness of PCPE Solutions

Once we select an appropriate set of control rules for PCPE, we need to determine whether the fitness of the solutions we obtain are heterogeneous. With this purpose, we have run some experiments over a set of benchmarks and different fitness functions, in order to collect statistical facts such as *Standard Deviation* and *Diameter*, that can help us to determine how different are the obtained solutions.

In these experiments, we use a set the same set $\mathcal{CS}$ of specialization strategies as in the experiments performed in Section 5.9. Also, we use some of the fitness functions defined in Appendix A and some of the benchmarks described in Appendix B. The set of benchmarks used in this experiment and the particular input queries used to specialize each program are shown in Table 6.1.

6.2.1 Heterogeneity of Solutions: Speedup

In Table 6.2 we can observe, for a number of benchmarks, the collected statistics when using Speedup as a fitness function. Remember that Speedup compares programs based on their time-efficiency, measuring run-time speedup w.r.t. the original program. As mentioned before, the number of solutions (shown in column *#Sols*) obtained by PCPE is tightly related to several factors, such as the number and kind of specialization strategies used, as well as the initial input queries used

Benchmark	#Sols	Fitness		St Dev	Diameter
		mfv	Mean		
example_pcpe	27	1.56	0.87	0.21	0.99
permute	70	3.84	1.15	0.48	3.15
nrev	255	0.99	0.66	0.15	0.51
advisor	14	2.97	1.31	0.67	1.99
relative	43	26.50	3.45	4.84	25.59
ssuply	31	11.50	1.84	1.82	10.49
transpose	154	2.75	0.87	0.30	2.13
overall	**87.4**	**7.15**	**1.45**	**1.21**	**6.40**

Table 6.2: PCPE Statistics over Different Benchmarks (SPEEDUP)

to specialize each program. For this particular experiment, PCPE generated a mean of 87 candidate solutions per benchmark.

In most cases we can observe that both the maximal fitness value (column *mfv*) and the mean fitness are over 1, meaning that a speedup is achieved when comparing the obtained solutions w.r.t. the original program. In some cases, the mean speedup is below 1, indicating that many of the solutions are bad and get a slowdown w.r.t. the original program.

Let us take *transpose*, for example. In this particular benchmark, we can see that most of the 154 final solutions are slower than the original program, meaning that it is easy to specialize this program with different specialization strategies and obtain a solution that runs slower than the original program. Note however, that the solution of maximal fitness obtained by PCPE is 2.75 faster than the original program.

In order to answer our initial question, i.e., whether does PCPE provide a wide range of solutions, the columns we are interested in looking at are *St Dev* and *Diameter*. *St Dev* stands for standard deviation, and measures how spread out the values in a data set are. *Diameter* measures the difference of fitness among (any of) the solution(s) of maximal fitness when compared to (any of) the solution(s) of minimal fitness. Note that many of the solutions found by PCPE can have the same fitness value. Values closer to 0 in *St Dev* would indicate that most solutions have a similar fitness value. However, the mean *St Dev* is

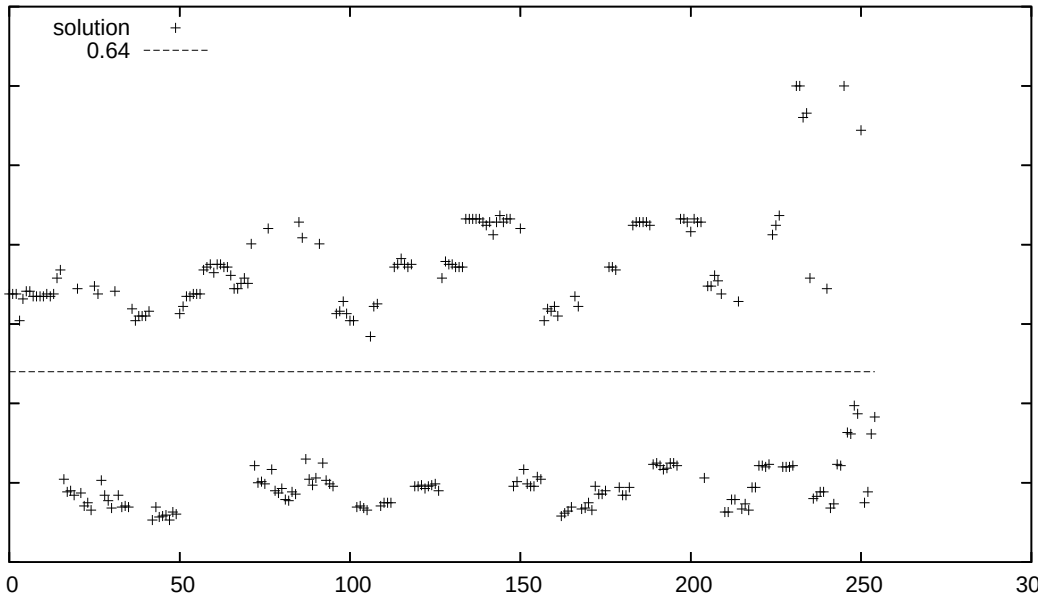

Figure 6.1: PCPE Solutions for `nrev` (SPEEDUP)

1.21, showing that in general solutions are spread out, i.e., they are different when compared with one another, even though very little static information is provided to the PCPE algorithm (as shown in Table 6.1). Regarding the *Diameter* column, we can observe that the mean diameter is 6.4, indicating that the solutions of maximal and minimal fitness are quite far away when considering their fitness values.

These claims can be better appreciated when looking at the fitness values of the different solutions in a graphical way. In Figure 6.1 we can observe, for benchmark `nrev`, how the fitness of all solutions are quite distributed across the mean value. Note that this benchmark is the one with the lowest *Standard Deviation* value, and with the highest number of versions obtained. Also, we can see that many solutions share the same fitness value, and that in some way they are grouped together, indicating that it should be possible to find ways to collapse those solutions into one, pruning in this way the search space. Similar conclusions can be obtained when looking at Figure 6.2, for the `permute` benchmark.

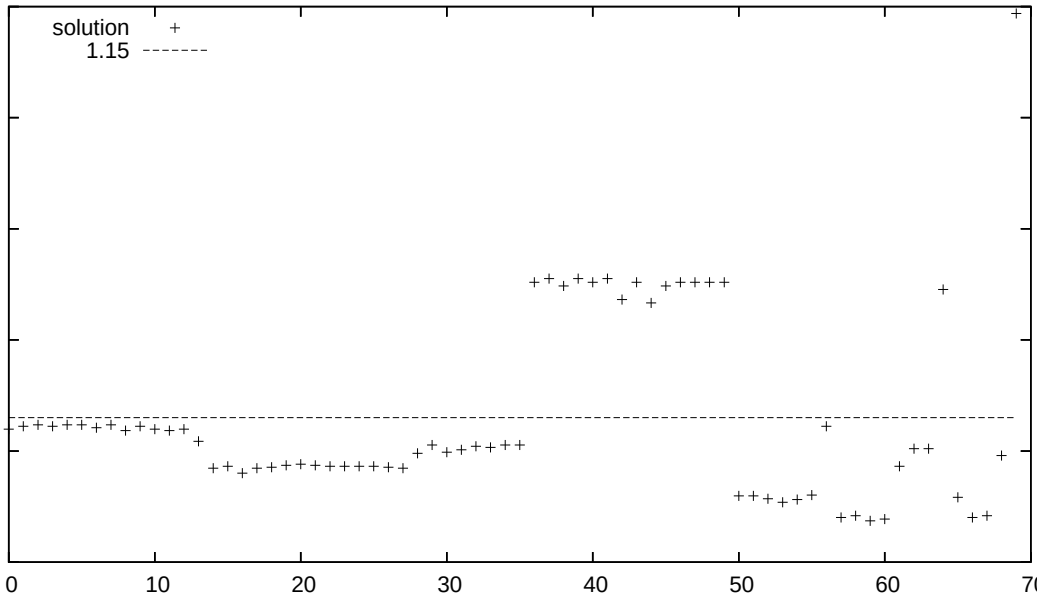

Figure 6.2: PCPE Solutions for `permute` (SPEEDUP)

6.2.2 Heterogeneity of Solutions: BYTECODE

In Table 6.3 we can observe the statistical values collected when running the same experiment as before, this time for the fitness function BYTECODE.

This table contains one extra column, *#mfv Sols*, denoting the number of solutions of maximal fitness value. Note that this column does not make any sense when using fitness function that measure time-efficiency (e.g. SPEEDUP or BALANCE), since it is virtually impossible to find two solutions sharing the fitness value, to the level of milliseconds, due to the noise introduced in time measurements (although graphically, it seems like many solutions have very similar fitness values, as we saw in Figures 6.1 and 6.2). In the case of fitness functions measuring space-efficiency, it is much easier to find solutions taking the same amount of bytes in disk or memory. Note that this does not mean that the solutions are the same, or even equivalent.

Since the number of versions obtained does not depend on the fitness function used, column *#Sols* has the same values as in Table 6.2. Note also that in most benchmarks there are many solutions sharing the same fitness value.

As in the case of SPEEDUP, we can see that in most benchmarks the mean

Benchmark	#Sols	#mfv Sols	Fitness		St Dev	Diameter
			mfv	Mean		
example_pcpe	27	1	1.22	0.82	0.19	0.82
permute	70	6	1.15	0.61	0.27	1.15
nrev	255	3	0.97	0.32	0.15	0.79
advisor	14	1	1.69	1.03	0.34	1.41
relative	61	2	1.17	0.67	0.25	1.04
ssuply	31	1	11.26	1.61	1.79	10.32
transpose	154	5	0.97	0.39	0.19	0.75
overall	**87.4**	**2.71**	**2.63**	**0.77**	**0.45**	**2.32**

Table 6.3: PCPE Statistics over Different Benchmarks (BYTECODE)

fitness value obtained is well under 1. This is very usual in the case of the BYTECODE fitness function, since partial evaluation normally focuses on time-efficiency, and tries to obtain as many specialized versions of a given predicate as possible (see Chapter 4), many times resulting in an explosion of the size of the code of the resulting program. Fortunately, for these benchmarks we observe that the solution of maximal fitness obtained by PCPE achieves a reduction of the bytecode size w.r.t. the original program.

It is worth to note that the standard deviation is lower than in the case of the SPEEDUP function. This could mean that the different solutions are more similar. By looking at Figure 6.3 (for benchmark `permute`) we can observe that there are many solutions grouped together sharing the same fitness value. This could indicate that, in this case, it can be safe to prune the search space, replacing all solutions in a group by a single one. This is not clear though when looking at Figure 6.4, for benchmark `relative`, where we can see very few groups sharing the same fitness value. We also show the graphics for `nrev`(Figure 6.6) and `transpose`(Figure 6.5) benchmarks, since these are the benchmarks having the lowest standard deviation values.

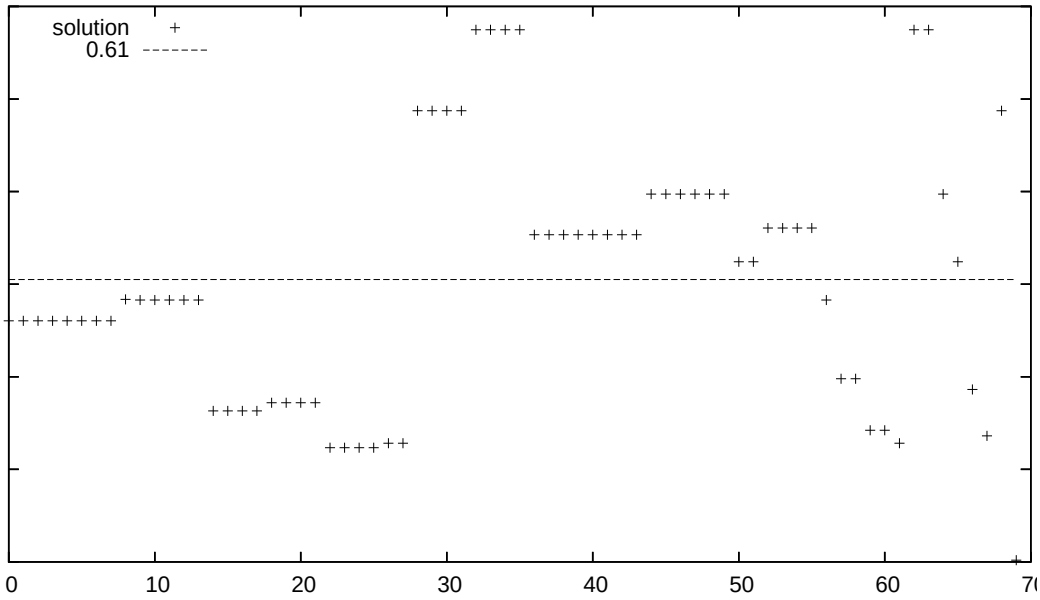

Figure 6.3: PCPE Solutions for `permute` (Bytecode)

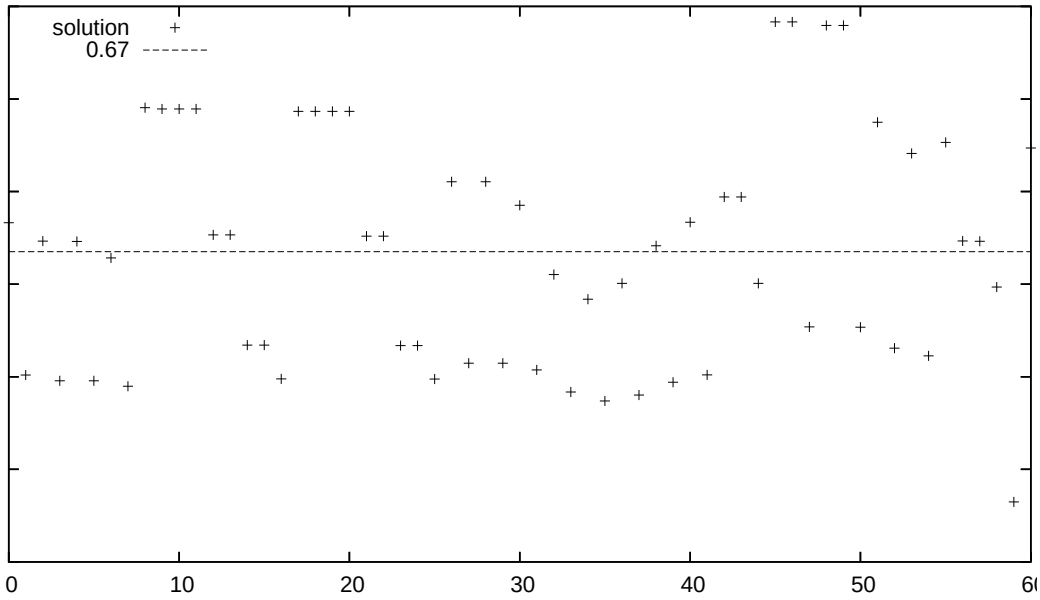

Figure 6.4: PCPE Solutions for `relative` (Bytecode)

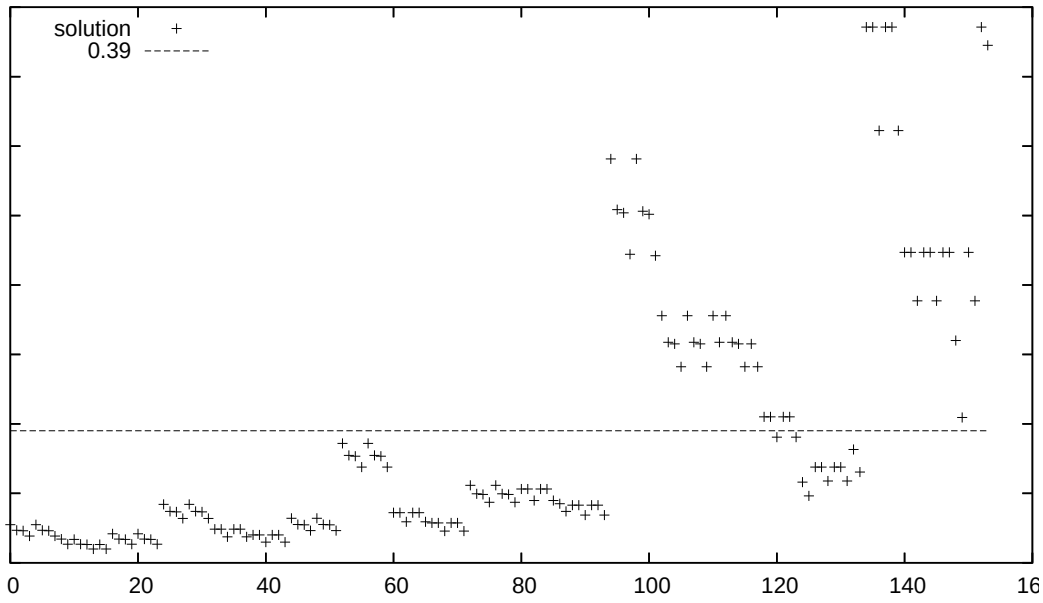

Figure 6.5: PCPE Solutions for `transpose` (Bytecode)

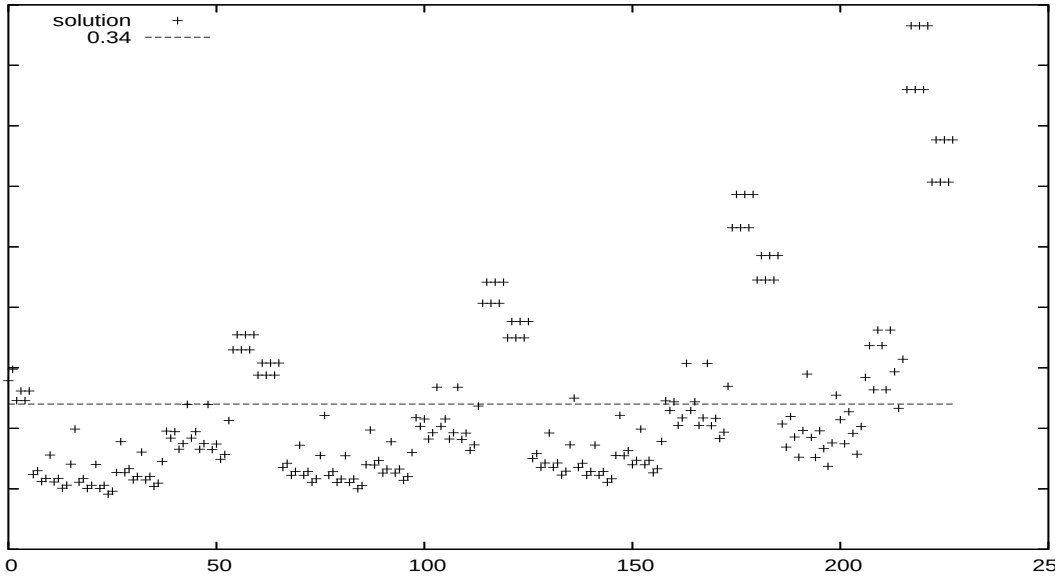

Figure 6.6: PCPE Solutions for `nrev` (Bytecode)

Benchmark	#Sols	Fitness		St Dev	Diameter
		Best	Mean		
example_pcpd	27	1.47	0.85	0.19	1.00
permute	70	1.21	0.80	0.26	1.09
nrev	255	0.99	0.45	0.12	0.69
advisor	14	2.18	1.10	0.32	1.30
relative	61	5.08	1.28	0.66	4.32
ssuply	31	11.83	1.73	1.88	10.84
transpose	154	1.22	0.53	0.20	0.88
overall	**87.4**	**3.42**	**0.96**	**0.51**	**2.87**

Table 6.4: PCPE Statistics over Different Benchmarks (BALANCE)

6.2.3 Heterogeneity of Solutions: BALANCE

Finally, in Table 6.4 we can see the values obtained when running the same experiment as above for the BALANCE fitness function. We also show, in a graphical way, the fitness values of all solutions for benchmarks `nrev`(Figure 6.7) and `permute`(Figure 6.8), since `nrev` is the benchmark with the lowest standard deviation value while `permute` is the one having the lowest diameter.

By looking at the values in the tables, and also at the figures, it can be deduced that PCPE is able to obtain several solutions having different fitness values. Most of these solutions are *hybrid*, i.e., they are not achievable by traditional partial evaluation.

6.3 Heterogeneity of PCPE Solutions: Highlights

After these experiments, we can try to empirically answer our first question posed early in this chapter: does PCPE provides a wide range of solutions? Based on these empirical results, we could say *yes*, PCPE is able to find several solutions, and their fitness values seem to be heterogeneous when compared to one another. One could argue that this set of benchmarks is very small to be representative. But it is also true that this set of benchmarks was probably the worst case for

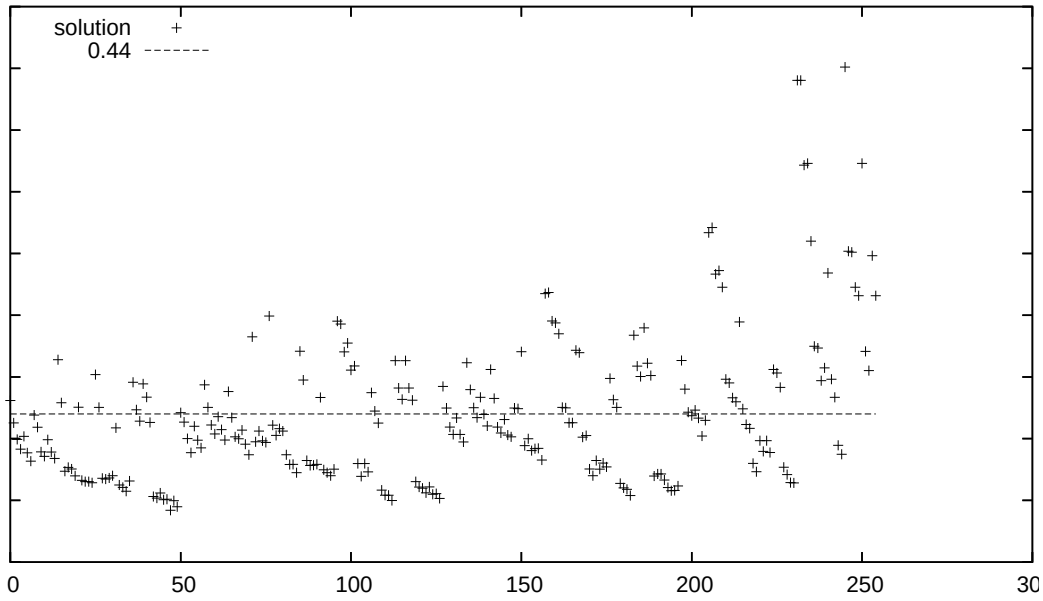

Figure 6.7: PCPE Solutions for `nrev` (BALANCE)

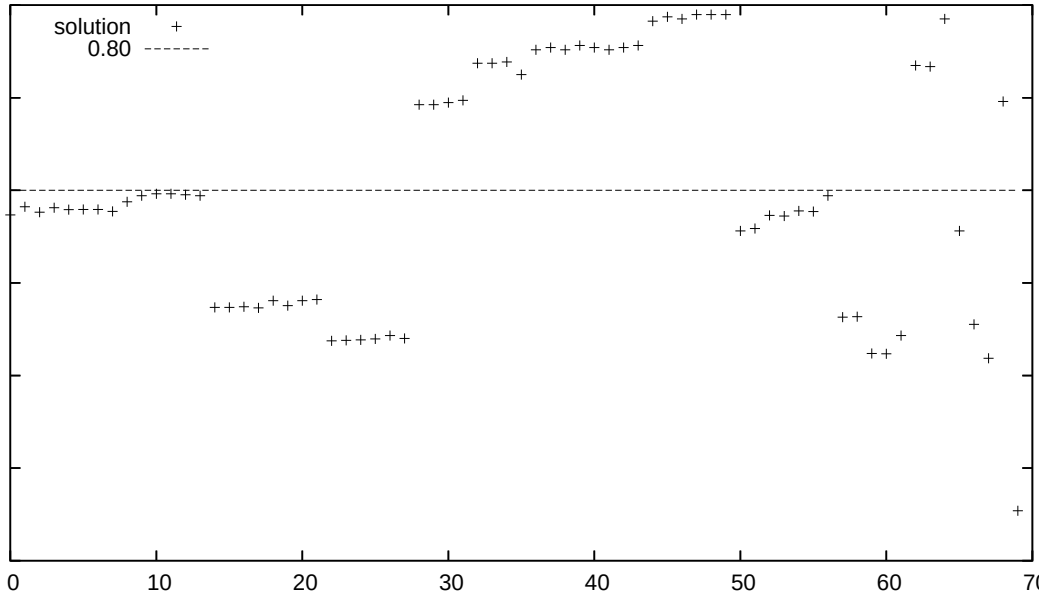

Figure 6.8: PCPE Solutions for `permute` (BALANCE)

carrying out these experiments, since most of them are small, and the amount of

static data provided is scarce, so the expected number and diversity of solutions found by PCPE is equally low.

This is an encouraging result, since it seems possible to find *hybrid* solutions (from among all of the solutions found by PCPE) that could have better fitness value than any *pure* solution (that could be found by traditional partial evaluation). We have seen some preliminary results in Chapter 5 where this happened. But in order to determine if there are many benchmarks and cases where this scenario happens, we have to make PCPE able to deal with more benchmarks, and more specific specialization queries, i.e., more static data. In the next part of this thesis we will describe several techniques for achieving this goal.

Part IV

Poly-Controlled Partial Evaluation In Practice

Chapter 7

The Search Space Explosion Problem

In this chapter we illustrate, by means of examples, one of the main problems of poly-controlled partial evaluation *when implemented as a search-based algorithm* (PCPE_{all}): the growth of the size of its search space.

In order to observe this phenomenon, in this chapter we will consider the program in Listing 7.1, which implements a naïve reverse algorithm:

Listing 7.1: The `rev/2` Example

```
:- module(_,[rev/2],[assertions]).
:- entry rev([_,_|L],R).

rev([],[]).
rev([H|L],R):-
        rev(L,Tmp),
        app(Tmp,[H],R).

app([],L,L).
app([X|Xs],Y,[X|Zs]):-
        app(Xs,Y,Zs).
```

In `CiaoPP`, the description of initial queries (i.e., the set of atoms of interest S in Algorithm 2 and Algorithm 3 presented in Chapter 5) is obtained by taking into account the set of predicates exported by the module, in this case {`rev/2`},

possibly qualified by means of *entry* declarations. For example, the *entry* declaration in Listing 7.1 is used to specialize the naïve reverse procedure for lists containing *at least* two elements.

In Section 7.1 we informally explain why the search space of PCPE can experience a (potentially) exponential blowup of its search space, then we introduce a trivial modification to Algorithm PCPE_{all} in order to reduce the size of the search space, and show how the search space of `nrev`, when using some particular control strategies, is drastically reduced due to this modification. Then, in Section 7.2 we run an experiment using the example in Listing 7.1 over different specialization strategies, to try and determine how the aggressiveness level of these specialization strategies affects the size of the search space of PCPE.

7.1 The Search Space of PCPE

Given that PCPE takes as input a set $\mathcal{CS} = \{CS_1, \ldots, CS_m\}$ of specialization strategies, for each configuration T_0 we can obtain m children configurations. The search-based PCPE algorithm (PCPE_{all}) presented in Chapter 5 generates all children for each configuration, so this represents a *potentially exponential* explosion in the size of the search space, and, in principle, it makes the algorithm impractical for dealing with realistic programs.

7.1.1 Eliminating Equivalent Sibling Configurations

Several optimizations can be done to the base search-based algorithm shown in Chapter 5, in order to deal with the problem of the growth of the search space of PCPE. A first obvious optimization is to *eliminate equivalent sibling configurations*.

Definition 7.1.1 (sibling configurations)**.** *Let T_i and T_j be two configurations. Then they are called* siblings *if there is a state T, and specialization strategies $CS_i \in \mathcal{CS}$ and $CS_j \in \mathcal{CS}$ s.t. $T \rightsquigarrow_{CS_i} T_i \wedge T \rightsquigarrow_{CS_j} T_j$.*

Definition 7.1.2 (equivalent sibling configurations)**.** *Let $T = \langle S, H \rangle$ be an intermediate configuration and let $A = TakeOne(S)$. Let $CS_1 = \langle G_1, U_1 \rangle$ be a specialization strategy s.t. $A'_1 = G_1(A, H)$ and $\tau_1 = U_1(P, A'_1)$. Let $CS_2 = \langle G_2, U_2 \rangle$*

be a specialization strategy s.t. $A'_2 = G_2(A, H)$ *and* $\tau_2 = U_2(P, A'_2)$. *Then* T_1 *and* T_2 *are* equivalent sibling configurations *iff* $T \leadsto_{CS_1} T_1$, $T \leadsto_{CS_2} T_2$, $A'_1 \approx A'_2$ *and* $\tau_1 \approx \tau_2$.

This optimization is easy to implement, not very costly to execute, and significantly reduces the size of the PCPE-tree. A simple example shows the magnitude of the reduction of the size of the search space of PCPE obtained by eliminating equivalent sibling configurations. Let us run PCPE over the `nrev` benchmark using a set $\mathcal{CS} = \{\langle G_1, U_1\rangle, \langle G_2, U_1\rangle\}$, where G_1 is based on homeomorphic embedding, G_2 abstracts away the value of all its arguments, and U_1 is a deterministic unfolding rule performing only leftmost derivation steps.

If we do not eliminate equivalent sibling configurations, then 28 final solutions are obtained by PCPE, and the corresponding PCPE-tree is shown in Figure 7.1. However, when eliminating equivalent sibling configurations, we drastically reduce the size of the PCPE-tree, obtaining only 4 final solutions, as shown in Figure 7.2.

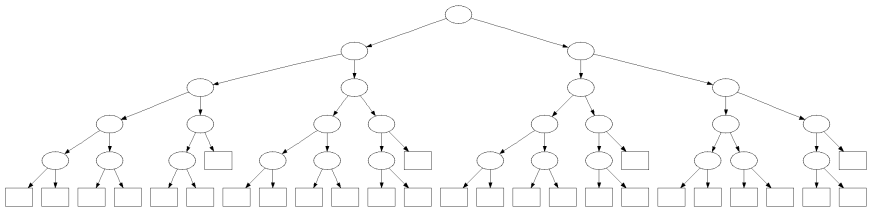

Figure 7.1: Search Space for `nrev` (With Equivalent Sibling Configurations)

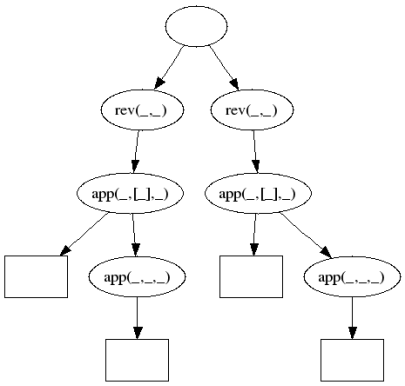

Figure 7.2: Search Space for `nrev` (Removing Equivalent Sibling Configurations)

Code	Global Control
G1	hom_emb
G2	dynamic

Table 7.1: Codes for Global Control Strategies

Code	Local Control
L1	unfolding(det) + comp_rule(leftmost) + unf_bra_fac(1)
L2	unfolding(one_step)
L3	unfolding(df_hom_emb_as) + comp_rule(local_emb) + unf_bra_fac(1)
L4	unfolding(df_hom_emb_as) + comp_rule(local_emb) + unf_bra_fac(0)

Table 7.2: Codes for Local Control Strategies

From this simple example it is clear that this optimization, although very simple and trivial, it greatly reduces the size of the search space. Moreover, as we will see later in this chapter, due to this simple optimization, in many cases the search space will not grow exponentially with the depth of the search tree.

7.2 Control Strategies and the Size of the Search Space

We now will show how different control strategies affect the size of the search space of PCPE. For this experiment, we use the global control rules listed in Table 7.1 and the local control rules listed in Table 7.2[1]. In these tables, we assign a unique code to each different global and local control rule. Note that in the case of the local control rule $L2$, the only relevant parameter is the unfolding strategy, since neither the value of the computation rule nor that of the branching factor affects the behaviour of the unfolding strategy.

We will run PCPE over the program `nrev`, as defined above, using all possible combinations of the control strategies described in Tables 7.1 and 7.2 s.t. either $|\mathcal{G}| > 1$ or $|\mathcal{U}| > 1$. We will also use different input queries, assigning a code to

[1]A local control rule is composed of the unfolding strategy, the computation rule, and the unfolding branching factor (all of them described in Chapter 3).

Code	Input query
Q1	rev(L,R)
Q2	rev([_|L],R)
Q3	rev([_,_|L],R)
Q4	rev([_,_,_|L],R)
Q5	rev([_,_,_,_|L],R)
Q6	rev([1|L],R)
Q7	rev([1,2|L],R)
Q8	rev([1,2,3|L],R)

Table 7.3: Codes for Input Queries

each of them, as described in Table 7.3.

Table 7.4 shows the number of candidate solutions generated by Algorithm PCPE_{all} (after eliminating equivalent sibling configurations), for each *input query*, and for each specialization strategies. From this table (and for this particular benchmark) we can extract the following interesting conjectures:

- When $\mathcal{G} = \{G2\}$, the number of candidate solutions *remains static*, no matter how many local control rules are being used, or what input query is being provided. This is due to the fact that this global control rule abstracts away all of the static data provided by the input query. Thus, the amount of information supplied by the input query does not affect the number of final solutions obtained by PCPE. Note that static data triggers further unfolding during local control. Thus, the absence of static data makes the behaviour of the different unfolding strategies very similar.

- When $\mathcal{G} = \{G1\}$, as the length of the list provided as input query grows, the number of candidate solutions computed also grows (in general):

  - This growing seems to be very slow when we use a set of two local control rules, and either $L1$ or $L3$ are one of them, since $L1$ and $L3$ are the most conservative local control rules.
  - $L4$ is the most aggresive local control rule of all four.
  - When $|\mathcal{U}| \geq 3$ the growing curve is more steep.

Code	Globals	Locals	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
CS1	G1	L1 + L2	1	1	2	3	4	1	2	3
CS2	G1	L1 + L3	1	2	2	2	2	2	2	2
CS3	G1	L1 + L4	3	9	9	9	9	15	15	15
CS4	G1	L2 + L3	1	2	3	4	5	2	3	4
CS5	G1	L2 + L4	3	9	21	33	45	15	27	39
CS6	G1	L3 + L4	3	6	6	6	6	6	6	6
CS7	G1	L1 + L2 + L3	1	2	4	6	8	2	4	6
CS8	G1	L1 + L2 + L4	3	9	24	39	54	15	30	45
CS9	G1	L1 + L3 + L4	3	16	16	16	16	26	26	36
CS10	G1	L2 + L3 + L4	3	16	39	62	85	26	49	72
CS11	G1	L1 + L2 + L3 + L4	3	16	42	68	94	26	52	78
CS12	G2	L1 + L2	1	1	1	1	1	1	1	1
CS13	G2	L1 + L3	1	1	1	1	1	1	1	1
CS14	G2	L1 + L4	3	6	6	6	6	6	6	6
CS15	G2	L2 + L3	1	1	1	1	1	1	1	1
CS16	G2	L2 + L4	3	6	6	6	6	6	6	6
CS17	G2	L3 + L4	3	6	6	6	6	6	6	6
CS18	G2	L1 + L2 + L3	1	1	1	1	1	1	1	1
CS19	G2	L1 + L2 + L4	3	6	6	6	6	6	6	6
CS20	G2	L1 + L3 + L4	3	6	6	6	6	6	6	6
CS21	G2	L2 + L3 + L4	3	6	6	6	6	6	6	6
CS22	G2	L1 + L2 + L3 + L4	3	6	6	6	6	6	6	6
CS23	G1 + G2	L1	2	4	4	4	4	6	10	18
CS24	G1 + G2	L2	2	4	6	8	10	6	14	30
CS25	G1 + G2	L3	2	4	4	4	4	6	10	18
CS26	G1 + G2	L4	2	9	9	9	9	17	32	62
CS27	G1 + G2	L1 + L2	2	4	8	12	16	6	22	62
CS28	G1 + G2	L1 + L3	2	6	6	6	6	10	18	34
CS29	G1 + G2	L1 + L4	6	48	48	48	48	129	246	480
CS30	G1 + G2	L2 + L3	2	6	10	14	18	10	30	78
CS31	G1 + G2	L2 + L4	6	48	108	168	228	129	480	1392
CS32	G1 + G2	L3 + L4	6	39	39	39	39	69	126	240
CS33	G1 + G2	L1 + L2 + L3	2	6	12	18	24	10	38	110
CS34	G1 + G2	L1 + L2 + L4	6	48	114	180	246	129	504	-
CS35	G1 + G2	L1 + L3 + L4	6	68	68	68	68	191	370	728
CS36	G1 + G2	L2 + L3 + L4	6	68	156	244	332	191	728	-
CS37	G1 + G2	L1 + L2 + L3 + L4	6	68	162	256	350	191	752	-

Table 7.4: Solutions Generated by PCPE for `rev` Benchmark

  - An important observation we can make is that, the more different the local control rules are, the more candidates we obtain. For example, $L4$ and $L2$ are very different, and as we can see in the row for $CS5$ the number of candidates quickly grows. However, $L3$ and $L4$ are quite similar, as they differ only in the branching factor. As a consequence, in $CS6$ we can observe that the number of solutions generated by PCPE remains low. This claim can be observed also in $CS30$ and $CS31$.

- When $\mathcal{G} = \{G1,\ G2\}$, and as before, the number of candidate solutions computed grows if the length of the list provided as input query also grows:

  - When $|\mathcal{U}\ | = 1$ the growing seems to be very slow when no actual elements are provided in the input query.
  - When $|\mathcal{U}\ | > 1$, and $L4 \in \mathcal{U}$, the number of final candidate solutions quickly grows, although in this case the growing seems to be linear (w.r.t. the amount of provided static data) too.
  - In a few cases we run out of memory, and this is indicated by a $-$ in the corresponding entry.

- In general, if the elements of the input list are static (as in the case of $Q6$, $Q7$ and $Q8$), the number of candidates solutions is higher than if these elements are unknown.

From this experiment we can see that the number of candidate specialized solutions generated by the poly-controlled partial evaluation algorithm depends greatly especially on the following factors:

- number and kind of control strategies used,
- aggressivity of the local control rules and
- amount of static data provided in the specialization queries

From this small example, it is clear that, in some cases (e.g. when having a big number of local and global control rules, or when using aggresive local control

rules), in order to be able to cope with realistic Prolog programs, it is mandatory to reduce the search space. This becomes evident when we take a look at the number of solutions produced for the different benchmarks of Appendix B, which is reflected in Table 7.5. As we can see, for many of them PCPE runs out of memory (indicated by a - in the table). We will present several techniques for pruning the search space in the upcoming chapters, and we will see how we can deal with most of these benchmarks.

Benchmark	**Query**	**Solutions**
example_pcpe	main(A,B,2,D)	27
permute	permute([1,2,3,4,5,6],L)	70
nrev	rev([_,_,_|L],R)	117
qsort	qsort([_,_,_|R],L)	-
qsortapp	qsort([1,2,3,4,5|L],R)	-
sublists	sublists([(2,3),(1,2),(3,8)|A],B,C)	-
freeof	list_freeof([_,_|L],p(p(A)))	794
sumexp	sumexp([_,_|_],4,_)	-
flattress	process(1+2+3,4+5+6,7/2,_,_|A],B)	-
mmatrix	mmultiply([[_,_|L],[_,_,_]|R],B,C)	-
datetime	add_dates([_,_|A],[[_,_],_|_],C)	-
advisor	what_to_do_today(first_of_may, _, _)	14
relative	relative(john, X)	61
ssupply	ssupply(_,_,_)	31
transpose	transpose([_,_,_,_,_,_,_,_,_],_,_)	154
rev_acc_type	rev(L,[],R)	30
depth	depth(member(X,[a,b,c,m,d,e,m,f,g,m,i,j]),D)	-
vanilla_db	solve_atom(a)	-
contains	contains([a,a,b],X)	-
ex_depth	solve([inboth(X,Y,Z)],0,Depth)	-
petri_object	unsafe(X,s(0),0,0,0)	-
match	match([a,a,b],String)	505
grammar	expression(n,[],String,[])	-

Table 7.5: Solutions Generated by PCPE for Different Benchmarks

Chapter 8

Heuristic Pruning

Although search-based PCPE is definitely appealing, its search space grows exponentially in the number of specialization strategies used. In order to deal with realistic programs, we need to *prune* the search space. This pruning can be performed using some heuristics—possibly losing solutions of maximal fitness—, or we can try to preserve solutions of maximal fitness, for instance, by means of branch and bound techniques. In this chapter we explore some pruning techniques of the first kind, and present a technique of the second kind in Chapter 9.

8.1 Predicate-Consistency Heuristics

In the algorithm PCPE_{all}, for any configuration T we apply all specialization strategies in $\mathcal{CS}$ to it, obtaining several children configurations. Rather than trying all possible specialization strategies, herein we propose to consider only those specialization strategies which are *consistent* with the choices previously taken in ancestor configurations.

The first notion of consistency we are going to consider is that we must use the *same control strategy for all atoms which correspond to the same predicate*. We will refer to configurations which satisfy this restriction as *predicate-consistent*, and as *predicate-inconsistent* to those which do not.

Definition 8.1.1 (predicate-consistent configuration)**.** *Given a configuration* $T = \langle S, H \rangle$*, we say that* T *is* predicate-consistent *iff* $\forall\ \langle A_1, A'_1, CS_1 \rangle, \langle A_2, A'_2, CS_2 \rangle \in H,\ pred(A_1) = pred(A_2) \Rightarrow CS_1 = CS_2$.

Note that this definition can be applied to both intermediate and final configurations. Thus, if a given intermediate configuration *Conf* is inconsistent, it will be pruned, i.e., it will not be pushed on *Confs*. By doing this we are pruning not only *Conf*, but also $solutions(Conf)$. This means that early prunings will achieve significant reductions of the search space.

Note that the predicate-consistent heuristic relies on storing the specialization strategy $CS = \langle G, U \rangle$ applied to each atom in the specialization history H.

Predicate-consistency will often significantly reduce the branching factor since handling of an atom A_i will become deterministic as soon as we have previously considered an atom for the same predicate in any configuration which is an ancestor of the current one in the search space, i.e., it is compulsory to use for A_i exactly the same specialization strategy used before. Though this simplification may look too restrictive at first sight, the intuition behind it is that, though it is often a good idea to allow using different specialization strategies for different predicates, it may be also the case that it is possible to obtain solutions of maximal fitness where we consistently use the same specialization strategy for all atoms of the same predicate. In other words, we believe that in the context of a given program, there may exists a specialization strategy which behaves well for all atoms which correspond to the same predicate. We thus propose to modify Algorithm PCPE_{all} such that only consistent configurations are further processed.

8.2 Mode-Consistency Heuristics

A possible improvement over predicate-consistency, in order to increase accuracy, is to define consistency at the level of *modes* for a predicate. This means that two calls to a predicate with similar *modes* (instantiation level in their arguments) must use the same specialization strategy, but not if they have different modes. In this sense, the more precise is the domain of modes, the less restrictive will be the simplifications made, thus producing a higher number of candidate solutions.

In order to check whether two atoms A and A' with $pred(A) = pred(A')$ have the same modes, we apply to them a function *modes* that abstracts their arguments one by one w.r.t. a given abstract domain [26].

Definition 8.2.1 (modes)**.** *Given an abstraction function α that abstracts an atom to an element of an abstract domain D, we define $modes_\alpha(p(t_1, \ldots, t_n))$ as*

Atom	*predicate*	$modes_{\alpha_{SD}}$	$modes_{\alpha_{SDL}}$	$modes_{\alpha_{SD}@depth(2)}$
$p(X, a)$	$p/2$	$p(D, S)$	$p(D, S)$	$p(D, a)$
$p(a, q(X, b), X)$	$p/3$	$p(S, D, D)$	$p(S, D, D)$	$p(a, q(D, b), D)$
$p(a, [], [a, X])$	$p/3$	$p(S, S, D)$	$p(S, S, L)$	$p(a, [], [a, D\|S])$
$p(a, q(b, X, r(Y, [])))$	$p/2$	$p(S, D)$	$p(S, D)$	$p(a, q(b, D,\ r(D, S)))$

Table 8.1: Abstraction of Calls Using Different Domains

$p(\alpha(t_1), \ldots, \alpha(t_n))$.

Then, we say that two atoms A and A' have the same modes under α iff $modes_\alpha(A) = modes_\alpha(A')$.

In a way, this is similar to the *binding types* used in the binding-time analysis (BTA) of *offline* partial evaluation [59]. In BTA, each argument of a predicate is given a *binding type* that provides some information about the instantiation state of an argument at specialization time.

The basic binding types in BTA are *static*, indicating that the argument is completely known at specialization time, and *dynamic*, indicating that the argument is possibly unknown at specialization time. Thus, in the same spirit, we define the α_{SD} *abstraction* as follows:

Definition 8.2.2 (α_{SD} abstraction)**.** *Given a term t, the α_{SD} abstraction over t is defined as follows:*

$$\alpha_{SD}(t) = \begin{cases} S & \text{if } vars(t) = \emptyset \\ D & \text{otherwise} \end{cases}$$

A more precise binding type can be defined by means of regular type declarations, and combined with basic binding types. For example, one can define types such as list skeletons. We can define the α_{SDL} *abstraction* as follows:

Definition 8.2.3 (α_{SDL} abstraction)**.** *Given a term t, the α_{SDL} abstraction over t is defined as follows:*

$$\alpha_{SDL}(t) = \begin{cases} S & \text{if } vars(t) = \emptyset \\ L & \text{if } t \text{ is bound to a list skeleton} \wedge vars(t) \neq \emptyset \\ D & \text{otherwise} \end{cases}$$

In addition to the abstractions described above, given an atom A, rather than applying the abstraction function directly to its arguments we can define

levels of depth at which *any* abstraction can be applied, in the spirit of the *depth-k* domains used in abstract interpretation of LP. For instance, given the atom $p(a, q(b, X, r(Y, [])))$, applying an abstraction α at level 1 would result in $p(a, q(\alpha(b), \alpha(X), \alpha(r(Y, []))))$, while applying such an abstraction at level 2 would produce $p(a, q(b, \alpha(X),\ r(\alpha(Y), \alpha([]))))$.

In Table 8.1 we can observe some examples of atoms, and how they are abstracted using the abstractions introduced above, including the α_{SD} *abstraction* at a depth level of 2 (column $\alpha_{SD}@depth(2)$).

We now provide a formal definition of consistent configurations w.r.t. the mode-consistent heuristics.

Definition 8.2.4 (mode-consistent configuration)**.** *Let α be an abstraction function. Then, given a configuration $T = \langle S, H\rangle$, we say that T is* mode-consistent *iff* $\forall\langle A_1, A_1', CS_1\rangle,\ \forall\langle A_2, A_2', CS_2\rangle \in H,\ (pred(A_1) = pred(A_2) \wedge modes_\alpha(A_1) = modes_\alpha(A_2)) \Rightarrow CS_1 = CS_2$.

8.3 An Heuristic-Based PCPE Algorithm

Algorithm 4 shows an Heuristic-based PCPE algorithm (H-PCPE). This algorithm is based on PCPE_{all}, with a slight modification that applies the heuristics presented in this chapter: the set *Candidates* of specialization strategies to be applied to the selected atom A_i (line 9) is now returned by a (new) function strategies. This function is depicted in Algorithm 5, and explained in detail below.

Algorithm 5 implements the function strategies, which takes cares of selecting an appropriate set of specialization strategies to be applied to a selected atom. The algorithm relies on a function called consistent (line 1), which given an atom A and a specialization history H, if $\exists\langle A_i, A_i', \langle G, U\rangle\rangle \in H$ s.t. A and A_i are (predicate|mode) consistent, it returns $\{\langle G, U\rangle\}$, otherwise it returns $\emptyset$.

8.4 Experimental Results

With the purpose of assessing the effectiveness of H-PCPE w.r.t. PCPE_{all}, we have performed a series of experiments using several benchmarks from Appendix B.

Algorithm 4 Heuristic-Based Poly-Controlled Partial Evaluation Algorithm (H-PCPE)

Input: Program P
Input: Set of atoms of interest S
Input: Set of specialization strategies $\mathcal{CS}$
Output: Set of partial evaluations *Sols*

1: $i = 0$
2: $H_0 = \emptyset$
3: $S_0 = S$
4: create($Confs$); $Confs$ = push($\langle S_0, H_0 \rangle$, $Confs$)
5: $Sols = \emptyset$
6: **repeat**
7: $\quad \langle S_i, H_i \rangle$ = pop($Confs$)
8: $\quad A_i = TakeOne(S_i)$
9: $\quad Candidates$ = strategies($A_i, H_i, \mathcal{CS}$)
10: $\quad$ **repeat**
11: $\quad\quad Candidates = Candidates - \{\langle G, U \rangle\}$
12: $\quad\quad A'_i = G(H_i, A_i)$
13: $\quad\quad \tau_i = U(P, A'_i)$
14: $\quad\quad H_{i+1} = H_i \cup \{\langle A_i, A'_i, \langle G, U \rangle\rangle\}$
15: $\quad\quad S_{i+1} = (S_i - \{A_i\}) \cup \{A \in leaves(\tau_i) \mid \forall \langle B, \_, \_ \rangle \in H_{i+1} \,.\, B \not\approx A\}$
16: $\quad\quad$ **if** S_{i+1}=$\emptyset$ **then**
17: $\quad\quad\quad Sols = Sols \cup \{H_{i+1}\}$
18: $\quad\quad$ **else**
19: $\quad\quad\quad$ push($\langle S_{i+1}, H_{i+1} \rangle$,$Confs$)
20: $\quad\quad$ **end if**
21: $\quad$ **until** $Candidates = \emptyset$
22: $\quad i = i + 1$
23: **until** empty_stack($Confs$)

In these experiments, we have used a set $\mathcal{CS}$ =$\{\langle G_1, U_1 \rangle, \langle G_1, U_2 \rangle, \langle G_2, U_1 \rangle, \langle G_2, U_2 \rangle\}$ of specialization strategies, where $G_1 = hom\_emb$ is an abstraction operator based on homeomorphic embedding, while $G_2 = dynamic$ abstracts away the value of

Algorithm 5 Selection of Specialization Strategies to be Applied to a Given Atom

Input: Atom A
Input: Specialization history H
Input: Set of specialization strategies $\mathcal{CS}$
Output: Set of specialization strategies $Cand$

1: $Strategy = \mathsf{consistent}(A, H)$
2: **if** $Strategy = \{\langle G, U\rangle\}$ **then**
3: $\quad Cand = Strategy$
4: **else**
5: $\quad Cand = \mathcal{CS}$
6: **end if**

all arguments. Also, $U_1 = hom\_emb\_aggr$ and $U_2 = hom\_emb\_cons$ are unfolding strategies which are both based on homeomorphic embedding for flagging possible non-termination (see [106] for more details). In both cases, non-leftmost unfolding is performed only when it is guaranteed to be *safe* (see [2]). However, the first one is more aggressive, whereas the second one is more conservative. More precisely, they differ in two ways:

1. the first one uses the *binding-insensitive* computation rule, whereas the second uses the *safe* computation rule of `CiaoPP`. The former is more aggressive, but it is only guaranteed to be correct in programs which are well typed. The second is correct for all programs.

2. more importantly, the second one only performs non-leftmost unfolding steps which are *determinate*, i.e., the selected atom must unify at most with one program rule. Note that this is important since it is well known that performing non-determinate non-leftmost unfolding can sometimes speedup the program but it often slowdowns the program. Thus, conservative rules tend not to perform this kind of unfolding steps.

Benchmark	PCPE			Best PE
	PCPE_{all}	H-PCPE		
		Predicate	**Mode**	
datetime	1.93	1.94	1.89	1.31
nrev	-	3.65	3.71	1.98
qsortapp	-	2.30	2.59	1.77
contains	4.16	4.16	4.21	3.15
grammar	8.41	8.42	8.43	4.71
groundunify_simple	-	3.02	-	2.95
liftsolve_app	1.17	1.16	1.19	1.17
match	1.59	1.58	1.58	1.09
transpose	2.46	2.46	2.45	2.46
Geom Mean	**2.60**	**2.60**	**2.60**	**1.98**

Table 8.2: Fitness for H-PCPE and Traditional PE

8.4.1 Benefits of Heuristic-Based PCPE

Table 8.2 compares the quality of the residual programs obtained by PCPE using the different heuristics presented in this chapter.

As a measure of the quality of programs we have used the fitness function Balance (see Appendix A). We believe that this fitness function is particularly interesting since it is resource-aware: both time-efficiency and size of the residual programs are taken into account. Thus, it is a useful fitness function in the context of pervasive and embedded systems, where it is important that the program does not exceed the storage capabilities of the device.

Since the fitness function Balance takes into account run-times of the residual programs, and there is always some noise associated to time measurement, times are taken as the arithmetic mean of ten consecutive runs. Nevertheless, when using Balance, is difficult to determine whether the selected solution is of maximal fitness or not, since fitness cannot be computed with full accuracy.

In order to make a fair comparison of PCPE w.r.t. traditional PE, we have run PE over all benchmarks with all four specialization strategies discussed above, looking for the specialization strategy which achieves the best overall fitness. For these particular benchmarks the best specialization strategy was $\langle G_1, U_2 \rangle$. Thus,

Benchmark	PCPE			Best PE
	PCPE_{all}	H-PCPE		
		Predicate	**Mode**	
datetime	21.2	3.6	5.0	21
nrev	-	2.2	10.1	14
qsortapp	-	3.3	10.6	32
contains	8.3	3.4	8.3	11
grammar	11.8	9.7	11.8	6
groundunify_simple	-	420.8	-	4
liftsolve_app	223.7	31.7	178.3	3
match	9.2	3.8	3.8	5
transpose	4.5	3.5	4.5	2
Geom Mean	**16.37**	**6.07**	**10.69**	**5.89**

Table 8.3: Normalized Size of Search Space w.r.t PE

we compare H-PCPE algorithm against PE using this particular specialization strategy.

In all tables in this chapter, the column PCPE_{all} represents the search-based PCPE from Chapter 5, where no pruning is performed. The two following columns, under the H-PCPE label, show the results of using heuristic pruning. The column `Predicate` presents the case where the predicate-consistency heuristics is used, and column `Mode` shows the results when the $modes_{\alpha_{SD}}$ heuristics is used.

In order to have a global view of the values in the different columns, in all tables we have included a row `Geom Mean` with the *geometric mean* of (part of) the values in the corresponding column. Since some columns do not have values for some benchmarks (because the corresponding algorithm has run out of memory), and in order to make comparisons meaningful, we compute the geometric mean only over those benchmarks which all algorithms can handle without problems, i.e., `nrev`, `qsortapp`, and `groundunify_simple` are not considered in this calculation. It should be noted that the specialization queries used in our runs contain a good amount of static information, thus causing PCPE_{all} to run out of memory (indicated by a $-$ in the table) for the programs mentioned above.

Benchmark	PCPE$_{all}$	H-PCPE	
		Predicate	**Mode**
datetime	105	6	12
nrev	-	4	24
qsortapp	-	6	18
contains	12	4	12
grammar	16	12	16
groundunify_simple	-	85	-
liftsolve_app	49	5	33
match	11	4	4
transpose	3	2	3
Geom Mean	**17.87**	**4.75**	**9.85**

Table 8.4: Number of Evaluations Performed

Fortunately, by using the heuristic-based pruning techniques presented in this chapter, H-PCPE has finished in most cases.

Several important conclusions can be drawn from Table 8.2. We can see that PCPE outperforms PE in most cases, achieving a mean fitness value of about 2.60 (vs 1.98 achieved by PE). The ratio PCPE/PE is around 1.31, which indicates that PCPE obtains residual programs which are about 31% better under the fitness function Balance than those achieved by traditional PE for this particular set of benchmarks. However, there are some cases where PCPE does not improve the results of PE because the program of maximal fitness is *pure* rather than hybrid. This happens, for example, in `transpose` and `liftsolve_app`.

Also, it can be seen that H-PCPE provides results whose fitness values are identical in most cases to the fitness obtained by PCPE$_{all}$. Although this could indicate that no pruning of the solution of maximal fitness is being done by H-PCPE, we cannot draw such conclusion since in many cases PCPE$_{all}$ does not finish, and thus, we have no reference of the solution of maximal fitness that could be obtained.

Benchmark	PCPE			Best PE
	PCPE_{all}	H-PCPE		
		Predicate	**Mode**	
datetime	1466	472	511	371
nrev	-	166	329	150
qsortapp	-	227	583	116
contains	532	326	417	267
grammar	626	458	553	290
groundunify_simple	-	4275	-	121
liftsolve_app	5924	418	1802	145
match	207	107	121	99
transpose	278	282	279	259
Geom Mean	**741.71**	**310.10**	**439.14**	**218.41**

Table 8.5: Analysis Times of H-PCPE (fitness = Balance)

8.4.2 Search Space of Heuristic-Based PCPE

The next question we address is whether the pruning techniques proposed can actually reduce the search space to levels which are comparable to those of traditional PE. Table 8.3 shows the *ratios* of the number of configurations generated by the different PCPE algorithms versus those generated by traditional PE. As can be seen, predicate-consistent H-PCPE achieves a significant reduction of the search space. It requires to explore, on average (only) 6.07 times the amount of configurations generated by PE, rather than 16.37 in the case of PCPE_{all}. Regarding mode-consistent H-PCPE, it can be seen that it generates around twice as many configurations as predicate-consistent H-PCPE, and it even runs out of memory for `groundunify_simple`. This, together with the fact that the maximal fitness value found using this heuristic are quite close to those obtained using predicate-consistent H-PCPE, allows concluding that predicate-consistent H-PCPE is preferable in practice.

Another important difference between PCPE and PE is that, since the former allows computing several candidate residual programs, we need to *evaluate* them in order to choose which one is the best of them. Table 8.4 shows the number

Benchmark	PCPE			Best PE
	PCPE_{all}	H-PCPE		
		Preds	**Modes**	
datetime	11972	654	1273	143
nrev	-	115	1025	58
qsortapp	-	672	2503	180
contains	710	209	72 0	82
grammar	1491	1094	1508	59
groundunify_simple	-	14456	-	13
liftsolve_app	4152	330	2665	30
match	266	93	108	32
transpose	55	34	53	15
Geom Mean	**957.39**	**232.04**	**525.64**	**42.43**

Table 8.6: Code Generation Times of H-PCPE (fitness = BALANCE)

of configurations which need to be evaluated, i.e., the number of times we need to compute or approximate fitness values. Since evaluations take place at the end of the algorithm, i.e., once all solutions have been generated, the number of evaluations coincides with the number of solutions generated by PCPE.

The results shown in the table indicate that H-PCPE requires an acceptable number of evaluations. For instance, predicate-consistent H-PCPE requires 4 times less evaluations than PCPE_{all}. We consider that these results are indeed quite promising and show that the search space and the number of evaluations required by PCPE are manageable, when the proposed pruning techniques are applied.

8.4.3 Time Cost of Heuristic-Based PCPE

Tables 8.5 to 8.8 show the time required by PCPE. All figures are in milliseconds. These experiments have been run using `Ciao` 1.13 over a 2.6 Linux kernel, on an Intel Pentium IV 3.4GHz processor, with 512Mb of RAM.

PCPE can be thought of as having three sequential phases, :

1. *analysis*, i.e., where the complete PCPE-tree is generated and explored,

Benchmark	PCPE$_{all}$	H-PCPE	
		Predicate	**Modes**
datetime	39704	2906	5016
nrev	-	1419	5135
qsortapp	-	2611	6404
contains	3218	1516	3200
grammar	4753	3670	4620
groundunify_simple	-	45329	-
liftsolve_app	20305	2445	13427
match	4202	1802	1777
transpose	1296	1035	1322
Geom Mean	**6375.71**	**2047.86**	**3643.59**

Table 8.7: Evaluation Times of H-PCPE (fitness = BALANCE)

2. *code generation*, i.e., where code is generated for all final configurations,

3. *evaluation*, i.e., where all specialized programs are assessed by using the corresponding fitness function.

Thus, in Table 8.5 we show the time spent in the analysis phase by both PCPE and PE, while in Table 8.6 we show the times spent in code generation by these approaches. When considering analysis and code generation times combined, H-PCPE performs around 4 times slower than PE, which is a rather reasonable cost. However, when compared to PE, PCPE requires an additional phase of evaluation in order to select the best candidate. The time required for evaluating candidates can vary a great deal from some fitness functions to others. In particular, if the fitness function measures time-efficiency, then in order to obtain accurate results several runs have to be performed. This can make PCPE much slower than PE, since the later does not perform this evaluation step.

This is the case with the fitness function used in these experiments. When evaluation time is taken into account, PCPE is an order or magnitude slower than PE. However, in many situations it can be argued that the cost of partial evaluation is not crucial since it takes place at compile-time. In fact, we believe that there can be a good number of cases where it is actually worthwhile to

Benchmark	PCPE			Best PE
	PCPE_{all}	H-PCPE		
		Predicate	**Mode**	
datetime	53142	4032	6800	514
nrev	-	1700	6489	208
qsortapp	-	3510	9491	296
contains	4460	2051	4337	349
grammar	6870	5222	6681	349
groundunify_simple	-	64060	-	134
liftsolve_app	30382	3193	17894	175
match	4675	2002	2006	131
transpose	1629	1351	1654	274
Geom Mean	**8498.84**	**2683.08**	**4764.77**	**267.37**

Table 8.8: Total Specialization Times of H-PCPE (fitness = BALANCE)

have the possibility of using a more powerful, though more expensive (but also completely automatic) framework for optimizing *relevant* code. This includes code which is going to be executed very often or code which has to be executed on devices with limited computing capabilities, as is the case in embedded systems.

Chapter 9

Branch and Bound Pruning

The main advantages of the heuristic-based pruning techniques shown in Chapter 8 are twofold: they are simple to implement, and they drastically reduce the search space of PCPE, thus reducing the specialization time and memory requirements and making H-PCPE able to cope with many more programs than PCPE_{all}. On the other hand, it is well known that heuristics can perform well for some cases and not so well for others. In the case of the heuristics just presented, this could mean that, in particular situations, solutions of maximal fitness may be lost. In this chapter we explore a different pruning technique based on *branch and bound* [70] (*BnB*), which guarantees the preservation of a solution of maximal fitness. Ideally, the new pruning technique should be applicable either in isolation or combined with the heuristic pruning already presented.

9.1 A Branch and Bound-Based Pruning

The basic idea of branch and bound-based pruning of the search space of PCPE is to store the fitness value of the best solution found so far, and prune away those configurations which are guaranteed not to improve the (temporary) solution of maximal fitness. We call this algorithm BnB-PCPE. Given an intermediate configuration T and a fitness function Fit, our aim is to be able to find an *estimated fitness* function EF_{Fit} that is an *upper bound* of the maximal fitness value of such configuration, i.e., $EF_{Fit}(T) \geq mfv_{Fit}(T)$. Then, if $EF_{Fit}(T)$ is smaller than the *actual* fitness value of the best solution found so far, we can safely prune T away.

In order to implement BnB-PCPE we need:

- to devise a mechanism for computing an *upper bound* EF_{Fit} of the maximal fitness value of any intermediate configuration.

- to traverse the PCPE-tree in a *depth-first fashion*, since this facilitates the finding of new final configurations, and its comparison with the current best fitness value. This will result in a more effective pruning, since newer best fitness values will make more probable to prune away non-promising branches. This traversal allow us to rapidly find a specialized solution (which will be pure, since Algorithm 3 always chooses control strategies in a fixed order), which automatically will become the best solution found so far. In addition, and as the search continues, further final configurations will be found which will possibly increase the fitness value of the best solution found so far.

- to decide how often we should evaluate candidates and try to prune. Clearly, if we evaluate often we will be able to prune more. However, evaluating introduces a non negligible cost. Thus, evaluating too often can make BnB-PCPE even slower than PCPE_{all}. As a result, in our implementation we do not evaluate every configuration. Instead, the implementation is parametric w.r.t. a *depth-level*. Those configurations which appear at a depth which is a multiple of the depth-level are evaluated. All others are not.

This pruning technique can be combined with the predicate- and mode-consistency heuristics presented in Chapter 8. In this case, we will obtain a solution with a fitness value which is guaranteed to be maximal among those final configurations which are consistent w.r.t. the abstraction used.

9.2 Estimating Fitness Values

As already mentioned, we need to obtain upper bounds on the maximal fitness values of intermediate configurations. The way we do it is tightly coupled to the fitness function being used. Remember that we assume that the fitness function returns values in the interval $[0, \infty)$ and that larger values are preferable to

smaller values. In our framework, we use different *resource-aware* fitness functions, described in Appendix A. In order to obtain the upper bounds over the different fitness functions, we will use the original program P_{orig} being specialized, and an empty program P_{empty} containing no body.

9.2.1 Estimated Bytecode (Memory) Fitness Function

In the case of Bytecode —or for any fitness function that takes the size of the resulting program as one factor (e.g. Memory)—, for any intermediate configuration $T{=}\langle S, H\rangle$ we can use the size of the resultants in the atoms in H in order to compute an upper bound of its maximal fitness value. In other words, we simply compile $\langle S, H\rangle$, i.e., we obtain an *incomplete* program out of the atoms contained in the set of already handled atoms H, and take the size of its bytecode in order to compute its estimated fitness value. We assume the existence of a function $Sizeb$ that returns the bytecode size (in disk) of a program P. Also, remember that the function $SP(T)$, defined in Chapter 5, takes a configuration T and computes its residual code (see Def. 5.2.8).

Definition 9.2.1 (estimated bytecode fitness function, $EF_{bytecode}$)**.** *Let T be an intermediate configuration. Then the* estimated bytecode *fitness of T is defined as*

$$EF_{bytecode}(T) = \frac{Sizeb(P_{orig}) - Sizeb(P_{empty})}{Sizeb(SP(T)) - Sizeb(P_{empty})}$$

The estimated memory fitness function is defined likewise, i.e., $EF_{memory}(T) = EF_{bytecode}(T)$.

Theorem 9.2.2. *Let T be an intermediate configuration. let FC be a final configuration reachable from T. Then, $EF_{bytecode}(T) \geq Fit_{bytecode}(SP(FC))$.*

Proof. By definition of the SP procedure, given two final configurations $FC_1 = \langle \emptyset, H_1\rangle$ and $FC_2 = \langle \emptyset, H_2\rangle$, if $H_1 \subseteq H_2$ then $Sizeb(SP(FC_1)) \leq Sizeb(SP(FC_2))$. Given an intermediate configuration $T = \langle S, H\rangle$, for any final configuration $FC = \langle \emptyset, H'\rangle$ s.t. FC is reachable from T, it holds that $H \subseteq H'$ since each PCPE-step always adds elements to H and never deletes elements from it. Therefore, $Sizeb(SP(T)) \leq Sizeb(SP(FC))$. It follows then
$EF_{bytecode}(T) = \frac{Sizeb(P_{orig}) - Sizeb(P_{empty})}{Sizeb(SP(T)) - Sizeb(P_{empty})} \geq \frac{Sizeb(P_{orig}) - Sizeb(P_{empty})}{Sizeb(SP(FC) - Sizeb(P_{empty})} = Fit_{bytecode}(SP(FC))$.
□

Thus, the estimated fitness value $EF_{bytecode}(T)$ is guaranteed to be an *upper bound* on $mfv_{bytecode}(T)$. In other words, if the size of the compiled incomplete program resulting from the current configuration T is already *larger* than the size of the current best solution, then we can safely prune away the current node and all of its descendants, since it will be impossible to obtain a program containing the already visited atoms which is smaller than the best solution found so far.

9.2.2 Estimated SPEEDUP Fitness Function

In the case of SPEEDUP, or for any fitness function that takes execution time of the residual program into consideration, we need to find an estimated speedup fitness function $EF_{speedup}$ s.t. for any intermediate configuration T, $EF_{speedup}(T)$ is an upper bound over its maximal fitness value, i.e., $EF_{speedup}(T) \geq mfv_{speedup}(T)$. In principle, and similarly to the case of BYTECODE, we could think of generating a program P_T out of an intermediate configuration $T = \langle S, H \rangle$, and running P_T to have an estimation of its execution time, and then compute its estimated speedup value. This poses a problem, since the partial evaluation algorithm is not devised in such a way that a consistent program can be obtained for any set of atoms. If such set of atoms is not *closed* [85], then the union of the partial evaluations for the atoms in the set $atoms(H)$ does not correspond to a self-contained program.

The solution we propose in this case is, given an intermediate configuration $\langle S, H \rangle$, to close it by using the original predicate definitions (from the program P being specialized) for all atoms in S, since we do not have a specialized version for them yet. This allows running the incomplete program.

Definition 9.2.3 (close)**.** *Let P be a program, let $T = \langle S, H \rangle$ be an intermediate configuration, let $P_H = SP(T)$, and let $P_{|S} \in P$ be a set of clauses s.t. $(\forall H_i \leftarrow B_1 \ldots B_n \in P_{|S}\ \exists A_j \in S\ .\ H_i \approx A_j) \wedge (\forall A_j \in S\ \exists H_i \leftarrow B_1 \ldots B_n \in P_{|S}\ .\ A_j \approx H_i)$. Then $close(T) = P_T = P_H \cup P_{|S}$.*

Unfortunately, given an intermediate configuration T, and a final configuration FC reachable from T, there is no relationship between $Time(close(T))$ and $Time(SP(FC))$.

To cope with this problem, we can make use of a profiler (see e.g. [32]) which allows defining *cost centers*. In our particular environment, we can take advantage of the profiler included in `Ciao` [90]. The profiler splits the total execution time

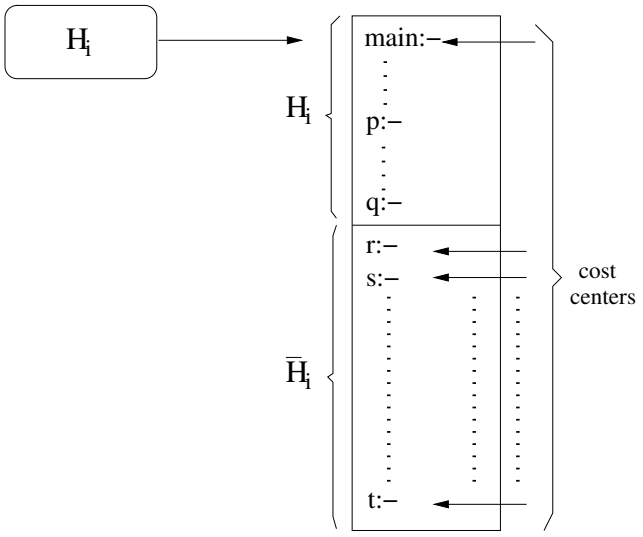

Figure 9.1: Profiling an Intermediate Configuration (SPEEDUP)

among the different predicates in the program. When a cost center is defined, it accumulates the execution time of all computations started from such predicate.

Thus, given a intermediate configuration $T = \langle S, H \rangle$, we close it obtaining a program P_T, and then we define cost centers for all predicates in $P_{|S}$ plus a cost center for the main entry (exported predicate) of the program (see Figure 9.1). Note that such exported atom must belong to H since it is the first atom handled by Algorithm PCPE_{all}. This way, when running the residual program, the time reported by the profiler for the main call of P_T will not include the time actually required for atoms which are not in H_i (represented by $\overline{H_i}$ in Figure 9.1). In other words, the profiler will split the time spent in predicates of P_H and $P_{|S}$, e.g, $Time(P_T) = Time(P_H) + Time(P_{|S})$. Then, we take $Time(P_H)$ as the estimated execution time of an intermediate configuration. Now we can defined an estimated speedup fitness function.

Definition 9.2.4 (estimated speedup fitness function, $EF_{speedup}$)**.** *Let T be an intermediate configuration. Let $P_T = close(T)$. Let $Time(P_T)$ be the execution time of P_T returned by a profiler s.t. $Time(P_T) = Time(P_H) + Time(P_{|S})$. Then, the* estimated speedup *fitness of T is defined as*

$EF_{speedup}(T) = \frac{Time(P_{orig})}{Time(P_H)}$.

Theorem 9.2.5. *Let T be an intermediate configuration. Let FC be a final con-*

figuration s.t. FC is reachable from T. Then $EF_{speedup}(T) \geq Fit_{speedup}(SP(FC))$.

Proof. Given $T = \langle S, H \rangle$ and $FC = \langle \emptyset, H' \rangle$ s.t. FC is reachable from T, it holds that $H \subseteq H'$ since each PCPE-step always adds elements to H and never deletes elements from it. Therefore, $Time(P_H) \leq Time(SP(FC))$, since we consider only the time spent in a subset of the predicates of the final program $SP(FC)$.

It follows then

$EF_{speedup}(T) = \frac{Time(P_{orig})}{Time(P_H)} \geq \frac{Time(P_{orig})}{Time(SP(FC))} = Fit_{speedup}(SP(FC))$. □

Thus, the execution time reported by the profiler for the main entry is, modulo timing noise, a lower bound of the execution time of any candidate solution reachable from the current configuration since we are using 0 as an estimate for the execution of all atoms in S_i. Therefore, if $EF_{speedup}$ is higher than the best time already found, again there is no point in further expanding the current configuration, and we can safely prune the corresponding branch.

9.2.3 Estimated BALANCE and BOUNDED_SIZE Fitness Functions

If we consider the BALANCE fitness function, we can simply estimate the upper bounds of SPEEDUP and BYTECODE as above, and apply the balance function to obtain an approximated fitness value which, as we need, is guaranteed to be an upper bound of the fitness value reachable from the current state.

Definition 9.2.6 (estimated balance fitness function)**.** *Let T be an intermediate configuration. Let $EF_{speedup}(T)$ be the estimated speedup fitness value of T and let $EF_{reduction}(T)$ be the estimated reduction fitness value of T. Then, the* estimated balance *fitness of T is defined as*

$EF_{balance}(T) = EF_{speedup}(T) \times EF_{reduction}(T)$

In the case of the BOUNDED_SIZE fitness function, we can simply take an intermediate configuration T, compute its estimated bytecode fitness value $EF_{bytecode}(T)$, and if this value is already higher than the bound of the size of the residual program, then we can prune this configuration away, otherwise we compute its estimated speedup fitness value $EF_{speedup}(T)$, and if this value is higher than the fitness value of the best solution found so far, then we can also prune T away.

Algorithm 6 BnB-Based PCPE Algorithm (BnB-PCPE)

Input: Program P

Input: Set of atoms of interest S

Input: Set of specialization strategies $\mathcal{CS}$

Output: A partial evaluation Sol

1: $i = 0$
2: $H_0 = \emptyset$
3: $S_0 = S$
4: create($Confs$); $Confs$ = push($\langle S_0, H_0 \rangle$, $Confs$)
5: $Sol = \epsilon$
6: $Fit = \phi$
7: **repeat**
8: $\quad \langle S_i, H_i \rangle$ = pop($Confs$)
9: $\quad A_i = Select(S_i)$
10: $\quad Candidates = \mathcal{CS}$
11: $\quad$ **repeat**
12: $\quad\quad Candidates = Candidates - \{\langle G, U \rangle\}$
13: $\quad\quad A'_i = G(H_i, A_i)$
14: $\quad\quad \tau_i = U(P, A'_i)$
15: $\quad\quad H_{i+1} = H_i \cup \{\langle A_i, A'_i, \langle G, U \rangle \rangle\}$
16: $\quad\quad S_{i+1} = (S_i - \{A_i\}) \cup \{A \in leaves(\tau_i) \mid \forall \langle B, \_, \_ \rangle \in H_{i+1} \; . \; B \not\approx A\}$
17: $\quad\quad$ **if** S_{i+1}=$\emptyset$ **then**
18: $\quad\quad\quad \langle Sol, Fit \rangle$ = best_solution($\langle Sol, Fit \rangle, H_{i+1}$)
19: $\quad\quad$ **else**
20: $\quad\quad\quad$ prune(i,Fit,$\langle S_{i+1}, H_{i+1} \rangle$,$Confs$)
21: $\quad\quad$ **end if**
22: $\quad$ **until** $Candidates = \emptyset$
23: $\quad i = i + 1$
24: **until** empty_stack($Confs$)

Algorithm 7 Selecting the (Temporary) Best Solution

Input: The evaluated current best solution $\langle Sol, SolFit \rangle$
Input: A newly found solution H
Output: The evaluated new best solution $\langle B, Bfit \rangle$

1: $Hfit = \mathsf{evaluate}(H)$
2: **if** $Sol = \epsilon$ **then**
3: $\quad \langle B, Bfit \rangle = \langle H, Hfit \rangle$
4: **else if** $HFit > SolFit$ **then**
5: $\quad \langle B, Bfit \rangle = \langle H, Hfit \rangle$
6: **else**
7: $\quad \langle B, Bfit \rangle = \langle Sol, SolFit \rangle$
8: **end if**

9.3 A Branch and Bound-Based PCPE Algorithm

Algorithm 6 shows a branch and bound-based PCPE algorithm (BnB-PCPE), based on the search-based PCPE algorithm PCPE_{all}. The main changes w.r.t. PCPE_{all} are:

- Algorithm 6 returns only one solution Sol.
- In line 18, best_solution takes care of evaluating the just found solution, and comparing it with the current best one (see Algorithm 7). Note that we always keep track of the current best fitness value Fit found so far.
- In line 20, pruning takes place through the procedure prune (see Algorithm 8).

Algorithm 7 evaluates a newly found solution, and compares it with the current best solution. The first solution found automatically becomes the best current solution. In order to determine that no solutions have been found so far we use the special symbol ϵ in line 5 of Algorithm 6. We also initialize the current best fitness Fit to a void value ϕ. In the actual implementation, there are several ways to represent this symbol. A possible way to do it is to represent Sol as a

Algorithm 8 Branch and Bound Pruning Procedure

Input: A depth level $Depth$

Input: A fitness value Fit

Input: An intermediate configuration $\langle S, H \rangle$

Input: A set of configurations $Confs$

Output: The evaluated new best solution $\langle B, BFit \rangle$

```
1: if (Depth % D) = 0 then
2:    UBound = evaluate(⟨S, H⟩)
3:    if UBound > Fit then
4:       push(⟨S, H⟩,Confs)
5:    end if
6: end if
```

singleton, i.e., a set with a single element. An empty set would mean that no solution has been found so far.

Finally, Algorithm 8 takes care of pruning intermediate configurations. This pruning is performed every D levels of depth, so this procedure immediately returns (and no pruning is performed) if the current depth $Depth$ is not a multiple of D. In this algorithm, % is the *modulo* operation, i.e., the remainder after a numerical division. Note that we evaluate the current intermediate configuration $\langle S, H \rangle$ by means of the function evaluate, whose implementation depends on the current fitness function, as explained in Section 9.2. $\langle S, H \rangle$ is pushed on $Confs$, i.e., is not pruned, only if the upper bound $UBound$ of its fitness value is higher than the current best fitness value. Otherwise is not pushed on $Confs$, i.e., such configuration will not be further explored.

Although functions best_solution and prune are quite simple and could be inlined in Algorithm 6, we represent them separately for the sake of readability. Also, in order to combine this branch and bound-based pruning with the heuristic pruning defined in Chapter 8, we can simply assign to the set of strategies $Candidates$ in line 10 the output of the function strategies defined in algorithm 5 of Chapter 8, i.e., line 10 would look like

$Candidates$ =strategies$(A_i, H_i, \mathcal{CS})$

The algorithm combining both pruning strategies is called HB-PCPE.

Benchmark	PCPE			Best PE
	BnB-PCPE	HB-PCPE		
		Predicate	**Mode**	
datetime	1.89	1.89	1.90	1.31
nrev	3.65	3.64	3.66	1.98
qsortapp	3.73	2.26	2.58	1.77
contains	4.17	4.17	4.16	3.15
grammar	8.42	8.44	8.44	4.71
groundunify_simple	2.96	2.95	2.97	2.95
liftsolve_app	1.18	1.17	1.17	1.17
match	1.61	1.61	1.60	1.09
transpose	2.44	2.44	2.46	2.46
Geom Mean	**2.85**	**2.74**	**2.69**	**2.05**

Table 9.1: Fitness of BnB-based PCPE and Traditional PE

9.4 Experimental Results

With the purpose of assessing the effectiveness of the pruning achieved by these branch and bound techniques, we compare BnB-PCPE and HB-PCPE against traditional PE through a series of experiments using the same set of benchmarks and specialization strategies used in the experiments of Chapter 8, i.e., $\mathcal{CS} = \{\langle G_1, U_1\rangle, \langle G_1, U_2\rangle, \langle G_2, U_1\rangle, \langle G_2, U_2\rangle\}$, where $G_1 = hom\_emb$ is an abstraction operator based on homeomorphic embedding, while $G_2 = dynamic$ abstracts away the value of all arguments. Also, $U_1 = hom\_emb\_aggr$ and $U_2 = hom\_emb\_cons$ are unfolding strategies which are both based on homeomorphic embedding s.t. U_1 is more aggressive, whereas U_2 is more conservative. The different PCPE algorithms are compared against PE using the combination of control strategies which achieves the best overall fitness. For these particular benchmarks the best combination is $\langle G_1, U_2\rangle$.

As a measure of the quality of programs we have used again the fitness function Balance, given that this is a *resource-aware* fitness function. Since Balance takes into account run-times of the residual programs, and there is always some noise associated to time measurement, times are taken as the arithmetic mean of ten consecutive runs.

Benchmark	PCPE			Best PE
	BnB-PCPE	HB-PCPE		
		Predicate	**Mode**	
datetime	16.1	3.0	4.4	21
nrev	7.7	2.1	4.9	14
qsortapp	8.4	2.5	4.9	32
contains	4.4	2.4	4.1	11
grammar	5.0	4.3	4.8	6
groundunify_simple	4.0	4.0	4.0	4
liftsolve_app	4.3	2.7	4.3	3
match	4.0	2.6	2.6	5
transpose	2.5	2.5	2.5	2
Geom Mean	**5.40**	**2.82**	**3.95**	**7.49**

Table 9.2: Normalized Size of Search Space

In all tables shown in this chapter, the column BnB-PCPE shows the results for the PCPE algorithm with branch and bound-based pruning. The next two columns, under HB-PCPE , show the cases where BnB is combined with two heuristics:

- column `Predicate` presents the case where the predicate-consistency heuristics is used, and
- column `Mode` shows the results where the modes-consistency heuristics is employed, using the abstraction $modes_{\alpha_{SD}}$.

In all our experiments we have set the parameter *depth-level* to 3. This means that those configurations which appear at depths 3, 6, 9... are evaluated and pruned if possible, but not those which are at depth levels not multiples of 3. We have empirically determined that 3 is an appropriate value for this parameter.

In order to have a global view of the values in the different columns, in all tables we have included a row `Geom Mean` with the *geometric mean* of the values in the corresponding column.

It should be noted that the specialization queries used in our runs contain a good amount of static information, and as we have seen in Chapter 7, this

Benchmark	BnB-PCPE	HB-PCPE	
		Predicate	**Mode**
datetime	149	18	31
nrev	52	7	25
qsortapp	111	18	50
contains	16	7	15
grammar	12	10	12
groundunify_simple	6	6	6
liftsolve_app	5	3	5
match	7	4	4
transpose	2	2	2
Geom Mean	**16.02**	**6.57**	**10.59**

Table 9.3: Number of Evaluations Performed

generates a large amount of intermediate configurations. Nevertheless, by using the branch-and-bound pruning, PCPE has always been able to finish.

9.4.1 Benefits of BPCPE

Table 9.1 shows the fitness values obtained by all approaches. We can see that PCPE outperforms PE in most cases, achieving a mean fitness value of about 2.76 (vs 2.05 achieved by PE). The ratio PCPE/PE is around 1.34, which indicates that PCPE obtains residual programs which are about 34% better under the fitness function Balance than those achieved by traditional PE for this particular set of benchmarks. However, there are some cases where PCPE does not improve the results of PE because the program of maximal fitness is *pure* rather than hybrid. This happens, for example, in `transpose` and `liftsolve_app`.

Also, it can be seen that when heuristic pruning is applied, it provides results whose fitness values are identical in most cases to the fitness obtained without pruning. An exception is `qsortapp`, where the fitness obtained by BnB-PCPE (3.73) is considerably larger than that obtained by Predicate-consistent HB-PCPE (2.26). In turn, this fitness is lower than that obtained by Modes-consistent HB-PCPE (2.58), which is the only case where the additional accuracy inherent to the mode-consistency heuristic seems to be of interest. Note that we could not

Benchmark	PCPE			Best PE
	BnB-PCPE	HB-PCPE		
		Predicate	**Mode**	
datetime	3084	1306	1407	371
nrev	1022	674	847	150
qsortapp	2289	799	1195	116
contains	1092	880	966	267
grammar	1354	1288	1280	290
groundunify_simple	819	789	778	121
liftsolve_app	793	726	750	145
match	670	614	625	99
transpose	822	808	810	259
Geom Mean	**1160.43**	**847.91**	**929.95**	**182.53**

Table 9.4: Analysis Times (fitness = BALANCE)

make this appreciation in Chapter 8, as PCPE_{all} did run out of memory for this benchmark. Another interesting observation is that BnB-based PCPE allows handling all considered benchmarks. This is because BnB reduces the search space significantly, thus avoiding *out of memory* problems.

9.4.2 Search Space of BnB-based PCPE

Table 9.2 shows the ratios of the number of configurations generated by the Bnb-based PCPE algorithms w.r.t. those generated by traditional PE. We can see that BnB-PCPE allows reducing the search space significantly. On average, we (only) need to explore 5.40 times as many configurations as in traditional PE.

Fortunately, the reduction of the search space achievable by BnB gets along quite well with the reduction achieved by the heuristics presented in Chapter 8. In fact, the best result in terms of search space is the combination of *BnB*+predicate-consistency, where on average it is only needed to explore less than *three times* (2.82) as many states as PE.

As we showed in Algorithm 6, when using branch and bound pruning, only one solution is obtained, and thus, there is no need for a post-evaluation step (as

Benchmark	PCPE			Best PE
	BnB-PCPE	HB-PCPE		
		Predicate	**Mode**	
datetime	100928	11298	18032	143
nrev	17351	2148	8542	58
qsortapp	71189	9543	34624	180
contains	5822	2117	5543	82
grammar	6117	5423	6092	59
groundunify_simple	2585	2691	2694	13
liftsolve_app	2249	1239	2246	30
match	2185	1239	1260	32
transpose	504	507	498	15
Geom Mean	**6732.68**	**2607.32**	**4449.60**	**48.00**

Table 9.5: Code Generation and Evaluation Times (fitness = BALANCE)

done in PCPE_{all}). However, in Table 9.3 we show the number of evaluations that are actually performed *during* the analysis phase of the BnB-PCPE algorithm, sometimes on final configurations, sometimes on incomplete configurations in order to prune them.

Again, the results shown in the table indicate that BnB-based PCPE requires an acceptable number of evaluations. One important point is that the number of evaluations performed when using BnB can sometimes be larger than when it is not. This is because in the case of BnB, also intermediate configurations can be evaluated. If little pruning is achieved, then more evaluations are needed. However, this will often be compensated by pruning, which reduces the number of configurations to be explored and of final solutions to be evaluated. All in all, we believe that the results shown in Table 9.2 and 9.3 are indeed quite promising and show that the search space and the number of evaluations required by PCPE are manageable, when the proposed pruning techniques are applied.

Benchmark	PCPE			Best PE
	BnB-PCPE	HB-PCPE		
		Predicate	**Mode**	
datetime	104012	12604	19439	514
nrev	18373	2822	9389	208
qsortapp	73479	10342	35819	296
contains	6914	2997	6509	349
grammar	7471	6711	7372	349
groundunify_simple	3404	3480	3472	134
liftsolve_app	3042	1965	2996	175
match	2855	1853	1885	131
transpose	1326	1315	1308	274
Geom Mean	**8695.90**	**3266.22**	**4362.44**	**267.37**

Table 9.6: Total Specialization Times (fitness = BALANCE)

9.4.3 Time Cost of BnB-based PCPE

In the BnB-based PCPE algorithm, the analysis, code generation, and evaluation phases are no longer sequential, but they are interleaved. In order to show the time spent in each of them, we discriminate the time spent in evaluating both intermediate and final configurations during analysis, and show this information in Table 9.5. Note that code generation is also included in this table since it is required to generate code prior to evaluating a configuration. Traditional PE, on the other hand, lacks of an evaluation phase, but still requires some code generation, so the corresponding time is shown on the rightmost column of Table 9.5. The rest of analysis time is shown in Table 9.4.

When considering analysis times, especially in the case of HB-PCPE, PCPE performs a bit over 4 times slower than PE. However, in order to perform pruning, several intermediate configurations must be evaluated, and this time is indicated in Table 9.5. The total specialization times are shown in Table 9.6. As can be seen, BnB-based PCPE is an order of magnitude slower than traditional PE. However, given that achieves specialized programs of much better quality, we believe that this time is no obstacle to use PCPE, especially when looking for resource-aware specialized programs.

Chapter 10

An Oracle-Based Poly-Controlled Partial Evaluation Approach

As seen in Chapter 7, when poly-controlled partial evaluation is implemented as a search-based algorithm, there is a large growth of the search space. This growth depends on factors such as the aggressiveness of the local control rules used, the number of specialization strategies chosen, and the amount of static data provided to the partial evaluation algorithm. In this chapter we investigate the possibility of using an *oracle* who decides which is the most promising specialization strategy for each call pattern based on the specialization results for such call patterns using the different strategies. All other branches in the tree are pruned away.

For example, let us take the motivating example of Section 5.1.1, defined in Listing 5.9. In Figure 10.1 we show its PCPE-tree. In this case, the best specialized program according to the fitness function Balance is obtained from the final configuration S_3. Thus, the oracle should be able to tell us: in T_1, after specializing using the two specialization strategies, the most promising state between T_2 and S_5 is T_2. Then, between T_3 and T_4, the latter is preferable. Then, T_6 is preferable to S_4. Finally, from T_6 we can only reach S_3.

The benefits of building such an oracle are twofold, since a single specialized program would be computed. First, we do not spend time generating multiple specialized programs. Second, as in the case of PE, we do not need an evaluation phase, which can be very costly. However, this approach can only be useful in practice if the oracle makes good decisions, since some of the specialized programs obtained by PCPE outperform PE, but others produce bad results.

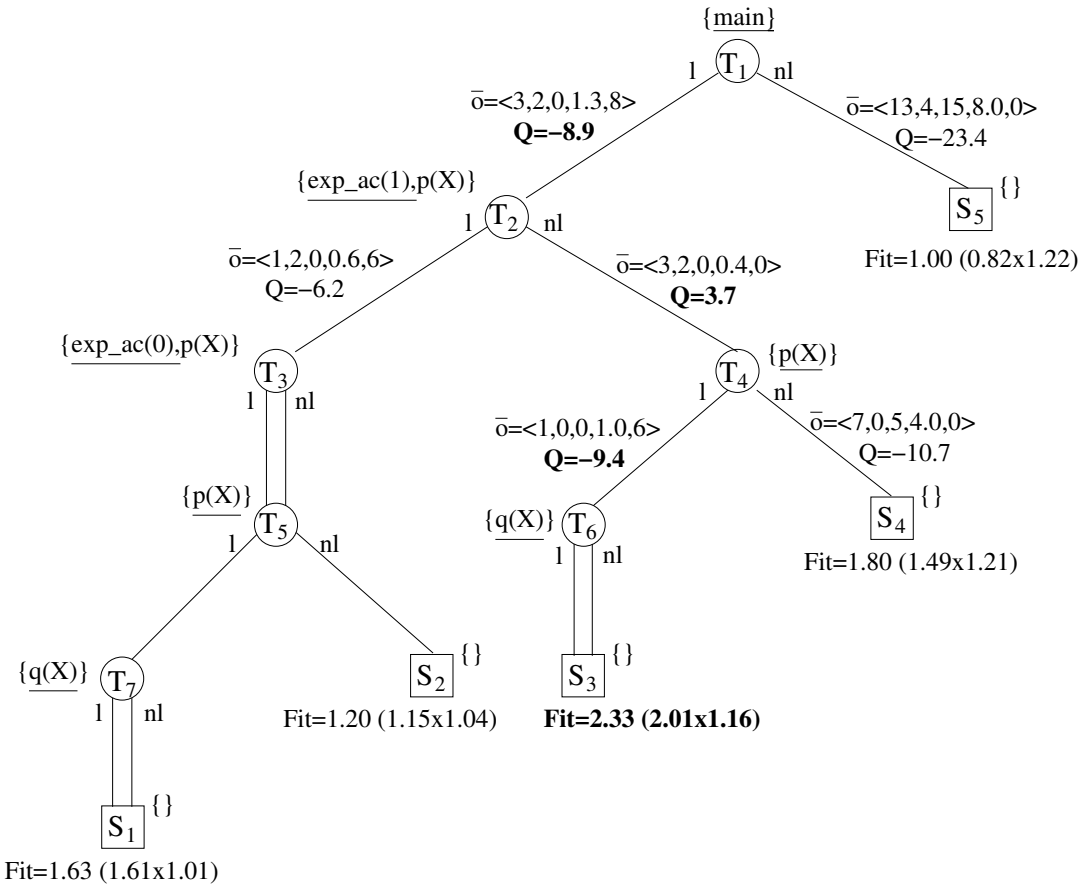

Figure 10.1: PCPE tree for program 5.9

We introduced an *empirical oracle* whose parameters are approximated from a set of training data. Such data is gathered from a set of calibrating examples and converted into a constraint logic program. Our experimental results show that specialization based on our empirical oracle introduces a constant overhead factor w.r.t. PE, while obtaining significantly better specialized programs.

10.1 Oracle-Based PCPE

The central idea behind *Oracle-based PCPE (*O-PCPE*)* is to traverse only one complete PCPE-path of a PCPE-tree. Note that, by doing so, we cannot always guarantee that we are traversing a PCPE-path of maximal fitness, since we need to compute all solutions to be sure we are making perfect decisions. For this,

given a configuration and a set of specialization strategies $\mathcal{CS}$, we generate all of its children using each specialization strategy in $\mathcal{CS}$, and choose the *most promising child* according to some *oracle* that uses information from the specialization process of each child.

Ideally, O-PCPE should always traverse PCPE-paths of maximal fitness. In practice, this depends on how appropriate the oracle function is w.r.t. the fitness function considered.

Definition 10.1.1 (oracle)**.** *Let P be a program. Let T be a configuration. Let $CS \in \mathcal{CS}$ be a specialization strategy s.t. $CS(T) = \tau$. An* oracle *is a function which receives as input T, τ and P and returns a number $Q \in \mathcal{Q}$. This is denoted $Q =$ $\mathsf{oracle}(T, \tau, P)$.*

A *perfect* oracle always obtains a solution of maximal fitness value.

Definition 10.1.2 (perfect oracle)**.** *Given a fitness function F, an oracle function* oracle *is* perfect *w.r.t. F if for any configuration T,*

$$\begin{aligned}(T \rightsquigarrow_{CS_i} T_i \wedge T \rightsquigarrow_{CS_j} T_j) \wedge \\ (\mathsf{oracle}(T, CS_i(T), P) \geq \mathsf{oracle}(T, CS_j(T), P)) \Rightarrow \\ mfv_F(T_i) \geq mfv_F(T_j)\end{aligned}$$

Finding a perfect oracle function will in general be impossible since the information available to the oracle is not quite enough in order to make perfect decisions. However, as our experimental results show, good results can be obtained without a perfect oracle function.

Since the oracle can rank two children with the same value, we impose an order on the generated children of a given configuration, by using a *sequence* of specialization strategies instead of a set. We can then use this order to break any possible tie.

We define now a function *mpchild*, which chooses a most promising child out of a sequence of children configurations. In case of a tie, *mpchild* selects the *first* configuration in the sequence having the highest Q value.

Definition 10.1.3 (mpchild)**.** *Let T be a configuration. Let $\mathcal{CS} = CS_1 : \ldots : CS_m$ be a* sequence *of specialization strategies. Let $\mathcal{T} = T_1 : \ldots : T_m$ with $T \rightsquigarrow_{CS_i} T_i$ be the children of T. Let $\mathcal{T}' = T_{i1} : \ldots : T_{in}$ be the maximal sub-sequence of $\mathcal{T}$ s.t.*

$\forall T_{ij} \in \mathcal{T}'$ oracle $(T, CS_{ij}(T), P) = Q$ *and* $\forall T_k \in \mathcal{T}$ oracle $(T, CS_k(T), P) \leq Q$. *Then* T_{i1} *is the* most promising child *of* T*, denoted* $T_{i1} = mpchild(T)$.

In O-PCPE, steps are deterministic: only the most promising child is expanded.

Definition 10.1.4 (O-PCPE-step)**.** *Let* T *be a configuration. Then an* O-PCPE-step *for* T *generates a new configuration* T' *s.t.* $T' = mpchild(T)$.

O-PCPE receives as input a program P, a set $\mathcal{A}$ of atoms describing the initial call patterns, a sequence $\mathcal{CS}$ of specialization strategies, and a selection function $TakeOne$. It starts by building an initial configuration $\langle \mathcal{A}, \emptyset \rangle$, and then performs a series of O-PCPE-steps until a final configuration $T = \langle \emptyset, H \rangle$ is reached, i.e., it traverses a complete PCPE-path, therefore generating only one specialized program $P' = SP(T)$.

10.2 An Empirical Oracle using a Linear Model

We now propose an oracle model which makes the problem of *empirically* determining an oracle function tractable. Furthermore, using this model, we obtain oracle functions which can be executed efficiently. This is important since during the specialization process the oracle is applied many times.

We propose to decompose the oracle function into two parts. The first one corresponds to computing the numerical value of a vector of *observables*, which should capture the relevant information about the specialization process. For this we use an auxiliary function quantify, which takes as input a configuration T, an SLD tree τ, and a program P and extracts the numeric value corresponding to each observable, denoted $\overline{o} =$ quantify(T, τ, P). The second part corresponds to the oracle function proper, which returns a numerical value as a function of the values of the observables.

10.2.1 Useful Observables for Resource-Aware Specialization

Since the oracle function will make its decisions based on the values of the observables, the practical success of O-PCPE has as prerequisite determining the

right set of observables for the considered fitness function. Those aspects of the specialization process which have a lot of impact on the quality of a specialized program should be considered. Otherwise, the oracle will not be able to make good decisions. In our case, as an example of a resource-aware specialization policy, we consider the fitness function BALANCE, which takes into account both the time and space efficiency of the specialized program P_T w.r.t. the original program P. Thus, the observables considered should somehow take these two factors into account.

Fig. 10.1 shows, next to each final configuration, the fitness value of the specialized program obtained from it, and also shows, in parentheses, the speedup and reduction values. In all our experiments we consider the following observables, where the first three ones are mostly related to time efficiency, and the last two to space efficiency:

D: The number of *derivation steps* that have been performed during unfolding and thus no longer need to be performed at runtime.

E: The number of *evaluation steps* that have been performed during unfolding. This indicates the number of calls to builtins and library predicates which have been evaluated [106] at specialization-time.

N: The number of atoms whose computation is replicated in several clauses as a result of *non-deterministic non-leftmost unfolding*. It is well-known that non-leftmost unfolding can increase the amount of computation required, by replicating the computation of atoms to the left of the selected one.

C: An estimation of the *growth of the residual code*, computed as a factor between the size (using a variation of the *term size* metrics [33]) of the specialized code for the selected atom A and the size of the original definition of the predicate $pred(A)$.

S: An estimation of the code size for the atoms added to S as a result of the last O-PCPE-step. Since no specialized code is available for these atoms, we use their original definition in P as an estimate of their size.

In most existing specialization strategies, which are focused on time efficiency, observables C and S are not explicitly handled and most heuristics aim at maximizing D and E while keeping N with the value zero. Observable S is an example

of information which is just partial when applying the oracle: in order to obtain a covered program, the code for the new atoms in S may in turn need including code for other atoms not yet covered. Perfect information can only be determined by actually expanding the PCPE-tree and observing it *a posteriori*.

Example 10.2.1. *Given the tree in Fig. 10.1, the value of the observables* $\langle D, E, N, C, S\rangle$ *which correspond to* $T_1 \leadsto_{\langle h,l\rangle} T_2$ *is* $\langle 3, 2, 0, 1.3, 8\rangle$*. This is because 3 derivation steps have been performed during unfolding of* `main(A,B,C)` *with the* l *rule (leftmost), and 2 calls to builtins have been evaluated. In this case, as well as in all configurations obtained by applying* l*, the value of* N *is 0, since* l *only performs leftmost derivation steps. The growth of the residual code w.r.t. the original definition of* `main/3` *is 1.3, and the estimation of the size of the code associated to the atoms* `exp_ac(1)` *and* `p(X)` *added to* S *is 8. Furthermore, in the case of* $T_1 \leadsto_{h,nl} S_5$*, the vector is* $\langle 13, 4, 15, 8.0, 0\rangle$ *because nl has performed 13 derivation steps and 4 evaluation steps. However, it replicates 15 atoms by doing non-deterministic non-leftmost unfolding. The growth of the residual code w.r.t. the original definition of* `main/3` *is 8.0 and* $S = 0$ *since no new atoms appear in the resultants.*

Measuring the Size of Terms

Various measurements can be used to determine the "size" of the terms appearing in the head and body of a given clause. We have implemented two measurements for terms, namely *term_count* and *term_size*.

Definition 10.2.2 (term_count)**.** *Let* t *be a term. Then* $term\_count(t) = 1$

As can be seen, *term_count* simply assigns 1 to any term, without considering its internal structure. A more precise measurement is *term_size*, adapted from [33].

Definition 10.2.3 (term_size)**.** *Let* t *be a term. Then*

$$term\_size(t) = \begin{cases} 1 & \text{if } var(t) \\ 1 + \sum_{i=1}^{n} term\_size(t_i) & \text{if } t = f(t_1, \ldots, t_n), n \geq 0 \end{cases}$$

Now we can define the size of a clause.

Definition 10.2.4 (size of a clause)**.** *Let $C = H \leftarrow B_1 \dots B_n$ be a clause. Then its size under a measurement m is defined as*

$$size_m(C) = size_m(H) + \sum_{i=1}^{n} size_m(B_i)$$

10.2.2 A Linear Model for the Oracle

In order to simplify our oracle model as much as possible, we will restrict ourselves to *linear* oracle functions.

Definition 10.2.5 (linear oracle function)**.** *Let $\overline{o} = \langle o_1, \dots, o_n \rangle$ be an observable vector. A* linear oracle function oracle *receives $\overline{o}$ as input and returns a numeric value $Q \in \mathbb{Q}$ which is defined as*

$$Q = \mathsf{oracle}(\overline{o}) = \sum_{i \in \{1,\dots,n\}} k_i \times o_i$$

where $\overline{k} = \langle k_1, \dots, k_n \rangle$ is a vector of oracle constants*, $k_i \in \mathbb{Q}$.*

To obtain a vector of oracle constants to be used with a given fitness function F, we build complete PCPE-trees, compute the mfv_F of all nodes, and then use this information as *training data*. For this, *O-constraints* are generated.

Definition 10.2.6 (O-constraint)**.** *Let F be a fitness function and T a configuration. Let T_1 and T_2 be two children of T s.t. $T \rightsquigarrow_{CS_1} T_1$ and $T \rightsquigarrow_{CS_2} T_2$. Let $\overline{o}_1 = \mathsf{quantify}(T, CS_1(T), P)$ and $\overline{o}_2 = \mathsf{quantify}(T, CS_2(T), P)$.*

Then the O-constraint *for the pair (T_1, T_2) is $\mathsf{oracle}(\overline{o}_1)$ $\mathcal{R}$ $\mathsf{oracle}(\overline{o}_2)$, where $mfv_F(T_1)$ $\mathcal{R}$ $mfv_F(T_2)$, $\mathcal{R} \in \{<, =, >\}$.*

Given a PCPE-tree *Tree*, we use $\mathcal{C}(\mathit{Tree})$ to denote the set of O-constraints which can be obtained from *Tree*. The cardinality of $\mathcal{C}(\mathit{Tree})$ is usually quite large: for each intermediate node T in *Tree* with p children we can build $\binom{p}{2}$ constraints for T. Thus, for a realistic tree *Tree* it is not possible to find a vector of oracle constants which allow satisfying all constraints in $\mathcal{C}(\mathit{Tree})$ simultaneously. There are several reasons for this. First, we have restricted ourselves to linear functions. It could be the case that there exists a non-linear oracle function which satisfies all constraints. However, the advantage of linear functions is that there exist

tools capable of handling them, whereas inferring non-linear functions is a rather complicated task. Second, as already mentioned, a perfect oracle function does not exist in general, since it has to make decisions based on partial information, i.e., without expanding the complete tree below the current node.

We can formulate the process of finding a vector of oracle constants as a *Maximum Constraint Satisfaction Problem* (MAX CSP) [23]. I.e., though the set of O-constraints is unsatisfiable, the goal then is to find a vector of oracle constants that maximizes the number of satisfied constraints in $\mathcal{C}(\mathit{Tree})$.

Unfortunately, the cardinality of $\mathcal{C}(\mathit{Tree})$ is large in general, and finding an optimal solution to this MAX CSP problem is quite costly. A simpler model results from collecting only (some of) the O-constraints occurring in a PCPE-path of maximal fitness.

Definition 10.2.7 (step-constraint)**.** *Let T be a configuration. Let $\mathcal{CS} = CS_1 : \ldots : CS_m$. Let $T_1 : \ldots : Tm$ be the children of T with $T \rightsquigarrow_{CS_i} T_i$. Let $T_i = mpchild(T)$. Then a* step-constraint *for T is*

$$\bigwedge_{j=1..m \wedge j \neq i} \mathsf{oracle}(\overline{o}_i) \geq \mathsf{oracle}(\overline{o}_j)$$

Example 10.2.8. *In the PCPE tree of Fig. 10.1, we have labeled some arcs with a vector $\overline{o} = \langle D, E, N, C, S \rangle$ containing the actual values the function* quantify *would return, and with the value Q computed for each vector $\overline{o}$ by using the empirical linear oracle function* oracle *we have obtained in our experiments.*

As can be seen in the figure, the PCPE-path traversed by O-PCPE *would be $T_1 \rightsquigarrow_{\langle h,l \rangle} T_2 \rightsquigarrow_{\langle h,nl \rangle} T_4 \rightsquigarrow_{\langle h,l \rangle} T_6 \rightsquigarrow S_3$, which coincides with the solution of maximal fitness value S_3, since $-8.9 \geq -23.4$, $3.7 \geq -6.2$, and $-9.4 \geq -10.7$.*

Also, an example of O-constraint generated for the pair (T_2, S_5), using the simplified linear model defined above, would be:
$3 \times D + 2 \times E + 0 \times N + 1.3 \times C + 8 \times S \geq 13 \times D + 4 \times E + 15 \times N + 8 \times C + 0 \times S$,
since $mfv_F(T_2) \geq mfv_F(S_5)$, F= Balance. This is also a step-constraint for T_1.

By collecting only those step-constraints in a PCPE-path of maximal fitness we have an instance of a MAX CSP that is more tractable than the original model that considered all O-constraints in a complete PCPE-tree.

In order to calibrate the oracle constants, we have used as *calibration benchmarks* those used in Chapters 8 and 9. The reason for this is that they are a representative set of PCPE examples for which it is possible to compute the complete PCPE tree. Basically, for each benchmark we collect a set $\mathcal{C}_j$ of step-constraints. Then, we enumerate all possible subsets $\mathcal{C}'_{ji} \subseteq \mathcal{C}_j$ and input each $\mathcal{C}'_{ji}$ to a Constraint Logic Programming solver[1], larger subsets first, until we find a maximal satisfiable subset $\mathcal{C}'_{ji}$ for each benchmark and its solution $\overline{k}_j = \langle k_{j1}, \ldots, k_{jn} \rangle$.

After collecting a set $\{\overline{k}_1, \ldots, \overline{k}_p\}$ of oracle constants, one for each of the calibration benchmarks, we normalize the value of each vector $\overline{k}_j$ by forcing the absolute value of the first constant k_{j1} (in our case corresponding to the observable D) to be 1 (written $|k_{j1}| = 1$). This is done by multiplying all constants $k_{j1}, \ldots, k_{jn}$ in each vector by $1/|k_{j1}|$. Note that this is a correct transformation since by multiplying a vector by a constant greater than zero, all constraints which were satisfied are again satisfied. Finally, the calibrated oracle constants result from computing the arithmetic mean over each normalized constant k_{ji}.

10.3 An Oracle-Based PCPE Algorithm

Algorithm 9 shows an *oracle-based* poly-controlled partial evaluation algorithm (*O-PCPE*). In each iteration of the algorithm, an atom A_i is selected from S_i (line 3). Then, we apply all specialization strategies to A_i (lines 8 and 9). This builds a SLD-tree τ for A'_i, a generalization of A_i as determined by G, using the unfolding rule U. The atoms in the residual code for A'_i, are collected by the function *leaves* (line 11). Those atoms in *leaves*(τ) which are not a variant of an atom handled in previous iterations of the algorithm are added to the set of atoms to be considered (S).

After building a child $\langle S, H \rangle$ (lines 10 and 11), the *oracle* evaluates it based on the values of a set of *implementation-dependant observables*, and a fitness function F, assigning a *quality value* Q to it (line 12).

All (evaluated) children configurations are stored in a set *Confs*, together with their quality value Q. Then, the most promising child is extracted by the function `best`, while the rest of configurations are discarded.

[1]We have used the **clpq** solver available in `Ciao` [17].

Algorithm 9 Oracle-based PCPE algorithm O-PCPE

Input: Program P

Input: A function oracle

Input: Set of atoms of interest S

Input: Sequence of specialization strategies $\mathcal{CS}$

Output: A partial evaluation for P and S, encoded by H_i

1: $i = 0$, $H_0 = \emptyset$, $S_0 = S$
2: **repeat**
3: $A_i = TakeOne(S_i)$
4: $Confs = \emptyset$
5: $Strategies = \mathcal{CS}$
6: **repeat**
7: $Strategies = Strategies - \{\langle G, U\rangle\}$
8: $A'_i = G(H_i, A_i)$
9: $\tau = U(P, A'_i)$
10: $H = H_i \cup \{\langle A_i, A'_i, \langle G, U\rangle\rangle\}$
11: $S = (S_i - \{A_i\}) \cup \{B \in leaves(\tau) \mid \forall\ \langle A, \_, \_\rangle \in H\ .\ B \not\approx A\}$
12: $Q = \mathsf{oracle}\ (\langle S, H\rangle, \tau)$
13: $Confs = Confs \cup \{(\langle S, H\rangle, Q_F)\}$
14: **until** $Strategies = \emptyset$
15: $\langle S_{i+1}, H_{i+1}\rangle = \mathsf{best}(Confs)$
16: $i = i + 1$
17: **until** $S_i = \emptyset$

10.4 Experimental Results

We have run a series of experiments in order to both evaluate the quality of the specialized programs obtained by means of O-PCPE and to compare the cost of this approach w.r.t. other specialization techniques.

We have used four specialization strategies, i.e., $\mathcal{CS} = CS_1 : CS_2 : CS_3 : CS_4$ with

$CS_1 = \langle hom\_emb, df\_hom\_emb\_as\rangle$,
$CS_2 = \langle hom\_emb, det\rangle$,

Benchmark	**LOC**	**Size**	PE (CS_1)	HB-PCPE	O-PCPE
contains	36	5549	2.76	3.63	3.76
datetime	282	17689	1.17	2.17	2.15
grammar	81	11831	8.10	15.21	15.38
liftsolve	97	7111	1.11	1.24	1.11
match	24	4781	0.54	0.95	0.98
nrev	25	4623	0.62	1.06	1.03
qsortapp	39	5390	0.55	1.04	1.06
transpose	33	5005	2.75	2.79	2.75
Geom Mean	**55.93**	**6863.38**	**1.40**	**2.14**	**2.12**

Table 10.1: Quality of Specialized Programs (Calibration Benchmarks)

$CS_3 = \langle dynamic, df\_hom\_emb\_as \rangle$, and
$CS_4 = \langle dynamic, det \rangle$.
More details on these abstraction operators and unfolding rules are found in Chapter 3.

The first phase of the experiments involves obtaining an empirical oracle function. For this, we have used the same benchmarks as in Chapters 8 and 9 and have executed HB-PCPE over them using the set $\mathcal{CS}$ mentioned above. During the second phase, we have used a set of benchmark programs *not* included in the set of calibrating benchmarks, since we are interested in knowing whether our oracle function obtains good results for arbitrary programs.

10.4.1 Using our Model within the Calibration Set

Table 10.1 compares the quality of the specialized programs obtained when specializing the calibration benchmarks using different specialization approaches. In order to compare specializations, we use the fitness value of the (best) solution found by each approach using the fitness function Balance.

Three different approaches to specialization have been considered. The first one is traditional partial evaluation (column PE (CS_1)). In order to make the comparison as fair as possible, we have run PE using all specialization strategies in $\mathcal{CS}$ over the calibration benchmarks, and chosen the one having the best

overall fitness value. In this case, the winning specialization strategy was CS_1. The second approach considered is the optimized search-based PCPE (HB-PCPE) presented in Chapter 9, which performs pruning of the search space based on a combination of heuristics and branch and bound techniques (column HB-PCPE). The third approach is the Oracle-based PCPE (O-PCPE) introduced in this chapter (column O-PCPE). For each benchmark, we show the number of lines of code (column **LOC**) and the size of the compiled bytecode (column **Size**).

As can be seen in Table 10.1, the overall fitness value of the solutions found by HB-PCPE (2.14) is around 53% better than that of traditional PE (1.40) for the best specialization strategy. This is mostly due to the fact that, as already mentioned, PE has been devised with time efficiency in mind, and can sometimes result in a code explosion. This is usually acceptable if only time-efficiency is considered (although sometimes can slow down programs due to cache miss effects), but it is harmful for resource-aware program specialization. Regarding O-PCPE, we can see that it obtains an overall fitness value (2.12) which is almost as good as the one obtained by HB-PCPE. It should be noted that balance takes time measurements into account, which may introduce some noise in the fitness values. This causes that, for some benchmarks (e.g. `contains` and `grammar`), O-PCPE seems to obtain higher fitness values than the ones obtained by HB-PCPE. To minimize this effect, fitness values in all tables result from averaging several runs of the generated solutions. This table indicates that O-PCPE behaves almost perfectly when specializing the calibration benchmarks. It is important to note that the aim of the empirical oracle function is not to be a perfect oracle function within the calibration benchmarks, but rather to behave well for arbitrary programs. In fact, it would be relatively simple to build an oracle which can reproduce PCPE-paths of maximal fitness for the calibration benchmarks, by simply memoizing PCPE-paths of maximal fitness previously selected. Instead, we build a general oracle function which is applicable to any program.

In addition to evaluating the benefits of O-PCPE, it is also important to evaluate the *cost* of PCPE, since this is the main drawback of the previous approaches to PCPE. In order to compare the cost of the different approaches to program specialization considered, Table 10.2 shows, for each approach, the number of configurations generated for each benchmark, and in the case of O-PCPE, some additional data. The row **Overall** shows the *geometric mean* computed over the

Benchmark	PE (CS_1)	HB-PCPE	O-PCPE		
			Confs	**Path**	**Ties**
datetime	21	75	16	6	0
nrev	14	56	8	4	0
qsortapp	32	142	12	4	0
contains	11	1515	27	10	0
grammar	6	1550	41	14	1
liftsolve_app	3	1188	70	30	0
match	5	49	8	4	0
transpose	2	21	9	4	0
Overall	**8.1**	**190.2**	**17.2**	**7.4**	**1.43%**

Table 10.2: Number of Configurations and Details on Specialization (Calibration Benchmarks)

different benchmarks. As can be seen, the number of configurations generated by HB-PCPE (190.2) is over 23 times larger than the number of configurations generated by traditional PE (8.1) even when all the techniques for pruning the search space of PCPE presented in Chapters 8 and 9 are applied. In fact, this is the main problem the search-based PCPE approach must cope with, since it not only affects time performance, but also the number of configurations is so large that in many cases HB-PCPE runs out of memory.

In contrast, the number of configurations generated by O-PCPE is bounded by the length of the PCPE path chosen by the oracle multiplied by the cardinality of $\mathcal{CS}$ (in our case, four). However, it is important to note that, as can be seen in the table, at least in our experiments, the number of configurations generated by O-PCPE (17.2), as shown in column **Confs** under O-PCPE , is slightly more than twice as many (instead of four times as many) configurations as traditional PE (8.1). There are several reasons for this. One is that the best solution under the Balance fitness function tends to have fewer predicates in the residual code, which implies that the path traversed is shorter. This can be observed in the column **Path**, which indicates the length of the PCPE-path which overall is smaller (7.4) than that of PE (8.1). Another reason for this is that, for efficiency, abstraction functions are applied first in the implementation, and then we unfold

Benchmark	PE (CS_1)	HB-PCPE				O-PCPE
	Tot	**Spec**	**Code**	**Eval**	**Tot**	**Tot**
datetime	**436**	352	723	1518	**2594**	**378**
nrev	**183**	143	193	770	**1106**	**129**
qsortapp	**341**	293	1131	2084	**3508**	**152**
contains	**258**	4418	13360	12893	**30671**	**272**
grammar	**205**	5030	77762	67339	**150131**	**327**
liftsolve_app	**190**	3476	7742	21227	**32445**	**357**
match	**136**	159	253	1146	**1558**	**110**
transpose	**130**	139	73	543	**755**	**129**
Geom Mean	**216.7**	**631.4**	**1482.7**	**3595.6**	**6038.7**	**206.7**

Table 10.3: Total Specialization Time in msecs. (Calibration Benchmarks)

those generalized atoms which are different. This means that if after abstracting we obtain two identical generalized atoms, only two children configurations will be generated, instead of four.

Another aspect which we consider important to evaluate in our experiments, is the number of times the oracle returns the same value for two children. This is shown in the column **Ties**. If this number were too high, it would probably indicate that the set of observables chosen does not convey enough information, and the possibility of choosing the wrong path would increase. Fortunately, for the calibration benchmarks this happens only in 1.43% of the total number of decisions taken.

Finally, Table 10.3 shows the total specialization times (in msecs) of the different approaches. In the case of HB-PCPE, the total time is split in the different phases of the specialization: time spent doing partial evaluation (**Spec**), time spent in code generation (**Code**), and time spent evaluating all generated solutions in order to choose the best one (**Eval**). All experiments have been run using `Ciao` 1.13 over a 2.6 Linux kernel, on a Pentium IV 3.4GHz CPU, with 512Mb of RAM. As noted in Chapter 9, if we only consider the time spent doing partial evaluation and code generation, then HB-PCPE takes an order of magnitude more time than traditional PE. The evaluation step required by HB-PCPE is the most costly step, and its cost depends on the particular fitness function being

Benchmark	PE (CS_1)			O-PCPE		
	Spec	**Code**	**Tot**	**Spec**	**Code**	**Tot**
datetime	270	166	**436**	277	101	**378**
nrev	118	65	**183**	118	11	**129**
qsortapp	141	200	**341**	134	18	**152**
contains	169	90	**258**	182	90	**272**
grammar	140	65	**205**	233	94	**327**
liftsolve_app	160	30	**190**	304	53	**357**
match	98	38	**136**	98	12	**110**
transpose	112	18	**130**	113	16	**129**
Geom Mean	**144.2**	**63.5**	**216.7**	**167.6**	**33.8**	**206.7**

Table 10.4: Specialization Time in msecs. (Calibration Benchmarks)

used. If time performance is being measured, then the generated solutions must be run several times with some sample queries in order to take significant time measurements.

Fortunately, O-PCPE is a greedy approach generating only one solution, and thus, the evaluation step is no longer needed. In Table 10.4 we show the total specialization times (in msecs) of PE and O-PCPE, split into time spent doing partial evaluation (**Spec**), and time spent in code generation (**Code**). We can see that the partial evaluation phase takes a bit longer than in the case of traditional PE (167.6 msecs. vs 144.2 msecs.), but on the other hand, the code generation phase takes less time since usually, as already discussed, PCPE paths traversed by O-PCPE are shorter than those traversed by traditional PE when using specialization strategy CS_1. Thus, the total time required by O-PCPE is competitive, in fact slightly smaller (206.7 vs 216.7), with the time required by PE when using CS_1.

10.4.2 Using our Model for Other Programs

Table 10.5 shows the benchmarks used in the second phase of our experiments. Some of these programs are actual libraries from existing Prolog systems, and most of them contain several hundred lines of source code, as shown in column

Benchmark	LOC	Size
analysis	343	39985
boyer	407	36619
browse	119	12579
credit	264	16932
exponential_peano	34	5639
groundunify_simple	78	10164
prolog_read	396	28300
qplan	397	37512
vanilla_db	110	13395
Mean	**238.67**	**22347.22**

Table 10.5: Benchmarks for experiments

LOC. In this table, column **Size** shows the size of the compiled bytecode of each benchmark.

Table 10.6 compares the quality of the specializations obtained in terms of the fitness value (using BALANCE) of the (best) solution found by each approach. In order to be as fair as possible, this time we compare both PCPE approaches (HB-PCPE and O-PCPE) using $\mathcal{CS}$ against traditional PE using *all* specialization strategies in $\mathcal{CS}$.

For each benchmark, we specify in bold the fitness value of the winning specialization strategy using PE. These values are not very high in several benchmarks. This is mainly because not much static data is available for such benchmarks. By looking at this table it seems that, at least for the BALANCE fitness function, there is no single specialization strategy which allows consistently obtaining good results. For instance, if we look at CS_1, which was the winning strategy for the calibration benchmarks, we see that in some cases it produces specialized programs that are considerably better than the original program—`groundunify_simple` (5.76) and `vanilla_db` (32.21)—while in most cases it obtains specialized programs that are worse than the original program (fitness values below 1).

An interesting case is `analysis`. The original program has 343 lines of code, the program obtained by CS_1 has over 38000 lines of code, and its compiled bytecode is over 5Gb. This is because *df_hom_emb_as* is an aggressive unfolding

Benchmark	PE				HB-PCPE	O-PCPE
	CS_1	CS_2	CS_3	CS_4		
analysis	0.0001	0.69	0.02	**1.03**	-	1.19
boyer	0.33	0.52	0.59	**0.99**	1.04	1.01
browse	0.78	1.76	2.21	**2.55**	2.65	2.57
credit	**1.64**	1.39	0.72	1.37	-	1.81
exponential_peano	0.55	0.86	0.57	**0.96**	0.95	0.86
groundunify_simple	**5.76**	4.63	0.27	1.01	-	6.04
prolog_read	0.08	0.10	0.94	**0.96**	-	5.09
qplan	0.84	0.84	1.02	**1.04**	-	0.99
vanilla_db	32.21	1.09	**32.39**	1.02	36.61	32.50
Geom Mean	**0.40**	**0.88**	**0.77**	**1.15**	**-**	**2.56**

Table 10.6: Quality of Specialized Programs

rule and it tries to unfold as much as possible, in this case resulting in code explosion, which is harmful in resource-aware program specialization. Indeed, the fitness value for this benchmark is so low that the geometric mean computed over all benchmarks but `analysis` is 1.14 (vs 0.40).

By looking at the overall results (row **Geom Mean**), it seems that the best specialization strategy for dealing with these benchmarks is CS_4. However, this is a quite conservative specialization strategy, generating a PE specialized program that is similar to the original program, without benefitting from the static information provided to the specializer. Thus, for many of the benchmarks, fitness results using CS_4 are close to 1, as observed in the table, with the exceptions of `browse` and `credit`.

On the other hand, HB-PCPE runs out of memory for several benchmarks, indicated with "−" in the table. As a result, we do not compute its geometric mean.

Finally, O-PCPE performs well in most cases, finding specialized programs that are, in average, 2.56 times better than the original one, as indicated in the **Geom Mean** row, and consistently similar or slightly better than the program obtained by the best PE, with a couple of exceptions:

- In the case of `exponential_peano`, the specialized program is worse than

Benchmark	PE $_{CS_1}$	PE $_{CS_4}$	O-PCPE		
			Confs	**Path**	**Ties**
analysis	334	54	227	77	2
boyer	83	32	44	15	7
browse	15	11	14	6	0
credit	28	25	82	32	3
exponential_peano	8	7	18	8	0
groundunify_simple	8	13	59	22	0
prolog_read	184	48	212	54	6
qplan	53	51	161	49	3
vanilla_db	7	4	19	10	0
Overall	**33.9**	**19.6**	**58.4**	**21.6**	**7.69 %**

Table 10.7: Number of Configurations and Details on Specialization

that achieved using CS_4. This is an indication that for this benchmark, our empirical oracle function has not made perfect decisions.

- The other exception is `prolog_read`, where the program obtained by O-PCPE is considerably better than any of the four programs obtained by PE, which in all cases have a fitness below 1.

The later is an indication that O-PCPE allows obtaining *hybrid* solutions which are not achievable using any of the specialization strategies in isolation, and which outperform the solutions of PE. Note also, that if we decide to use PE with several specialization strategies, we would again need to introduce an evaluation step much in the same way as in HB-PCPE and which is not needed in O-PCPE. Finally, it is worth mentioning that O-PCPE outperforms PE using any of the four specialization strategies in isolation by a factor of 2.22 or higher.

Regarding the cost of O-PCPE, Table 10.7 shows the number of configurations generated by PE and O-PCPE for each benchmark, and in the case of O-PCPE, some additional data. We no longer include HB-PCPE, since as already seen in Table 10.6, it runs out of memory in several of the benchmarks and it is an alternative only when the quality of the specialized program is of much importance, and the PCPE tree has a moderate size.

Benchmark	PE $_{CS_1}$	PE $_{CS_4}$	O-PCPE			
	Tot	**Tot**	**Spec**	**Ora**	**Code**	**Tot**
analysis	**38219**	**721**	13612	1254	339	**13951**
boyer	**11374**	**14748**	11037	25	203	**11240**
browse	**658**	**915**	623	4	50	**673**
credit	**399**	**429**	492	26	161	**653**
exponential_peano	**139**	**181**	110	2	23	**133**
groundunify_simple	**158**	**238**	208	10	52	**260**
prolog_read	**2728**	**711**	1839	438	616	**2455**
qplan	**811**	**1228**	642	81	447	**1089**
vanilla_db	**12246**	**227**	15675	7	3714	**19389**
Geom Mean	**1597.5**	**690.0**	**1391.8**	**28.5**	**206.3**	**1708.5**

Table 10.8: Specialization Time

The row **Overall** shows the *geometric mean* computed over the different benchmarks, except for the column **Ties**, which is discussed below. In order not to complicate the presentation too much, the comparison is against PE using CS_1 and CS_4 only, denoted PE_{CS_1} and PE_{CS_4} respectively, since the former is the most aggressive specialization strategy (and the winning strategy in the calibration bencharks), while the latter is the most conservative one (and also the one obtaining the best fitness values in Table 10.6).

It can be seen in the table that the number of configurations generated by O-PCPE (58.4), as shown in column **Confs** under O-PCPE, is slightly less than twice as many configurations as PE_{CS_1} (33.9), and three times as many configurations as PE_{CS_4} (19.6). Regarding the lenght of the PCPE-path (column **Path**) we can see that the overall of O-PCPE is smaller (21.6) than that of PE_{CS_1} (33.9) and quite similar to that of PE_{CS_4} (19.6). Column **Ties** shows the number of times the oracle returns the same value for two children. For these benchmarks, this happens only 7.69% of the total number of decisions taken.

Finally, Table 10.8 shows the total specialization times (in msecs) of both PE (using CS_1 and CS_4) and O-PCPE. In the case of O-PCPE, total specialization time (in columns **Tot**) is split into time spent doing partial evaluation (column **Spec**), time spent during code generation (column **Code**), and the time spent by

the oracle when selecting the most promising child configuration (column **Ora**).

We can see that the total specialization time of O-PCPE (1708.5) is quite similar to that of PE_{CS_1} (1597.5), and 2.47 times higher than that of PE_{CS_4} (690.0). These times are consistent with the number of configurations which need to be generated in the different approaches, plus the cost of code generation. An important point to mention is that the cost of PCPE represents a constant overhead factor w.r.t. PE. Such factor is directly proportional to the cardinality of $\mathcal{CS}$. For aggressive strategies, such as CS_1, the cost of PCPE is quite close to that of PE.

Also, we can see that the time spent by the oracle (28.5) is negligible when compared to the total specialization time (1708.5).

Part V

Poly-Controlled Partial Evaluation: Implementation

Chapter 11

Guidelines for the Use of PCPE

11.1 Integration of Poly-Controlled Partial Evaluation into CiaoPP

All of the work described in this thesis has been implemented in the program development system Ciao [17], and integrated into CiaoPP [20, 55, 19], the preprocessor of Ciao.

Ciao is a multi-paradigm programming system, allowing programming in logic, constraint, and functional styles (as well as a particular form of object-oriented programming). At the heart of Ciao is an efficient logic programming-based kernel language. This allows the use of the very large body of approximation domains, inference techniques, and tools for abstract interpretation-based semantic analysis which have been developed to a powerful and mature level in this area (see, e.g., [91, 22, 45, 18, 29, 54, 55] and their references). These techniques and systems can approximate at compile-time, always safely, and with a significant degree of precision, a wide range of properties which is much richer than, for example, traditional types. This includes data structure shape (including pointer sharing), independence, storage reuse, bounds on data structure sizes and other operational variable instantiation properties, as well as procedure-level properties such as determinacy, termination, non-failure, and bounds on resource consumption (time or space cost).

CiaoPP is a standalone preprocessor to the standard clause-level compiler. It performs source-to-source transformations. The input to CiaoPP are logic

programs (optionally with assertions and syntactic extensions). The output are *error/warning messages* plus the *transformed logic program*, with:

- Results of analysis (as assertions).
- Results of static checking of assertions.
- Assertion run-time checking code.
- Optimizations (specialization, parallelization, etc.)

By design, `CiaoPP` is a generic tool that can be easily customized to different programming systems and dialects and allows the integration of additional analyses in a simple way.

11.2 A Poly-Controlled Partial Evaluation Session Example

A `CiaoPP` session consists in the preprocessing of a file. The session is governed by a menu, where the user can choose the kind of preprocessing to be done to a file from among several analyses and program transformations available.

Clicking on the  icon in the buffer containing the file to be preprocessed, displays the initial menu, which will look (depending on the options available in the current `CiaoPP` version) something like the "Preprocessor Option Browser" shown in Figure 11.1.

Except for the first and last lines, which refer to loading or saving a menu configuration (a predetermined set of selected values for the different menu options), each line corresponds to an option the user can select, each having several possible values.

The preprocessing available in `CiaoPP` is located under the `Select Action Group` dropdown menu. The user can select either

- analysis (`analyze`) or
- assertion checking (`check_assertions`) or

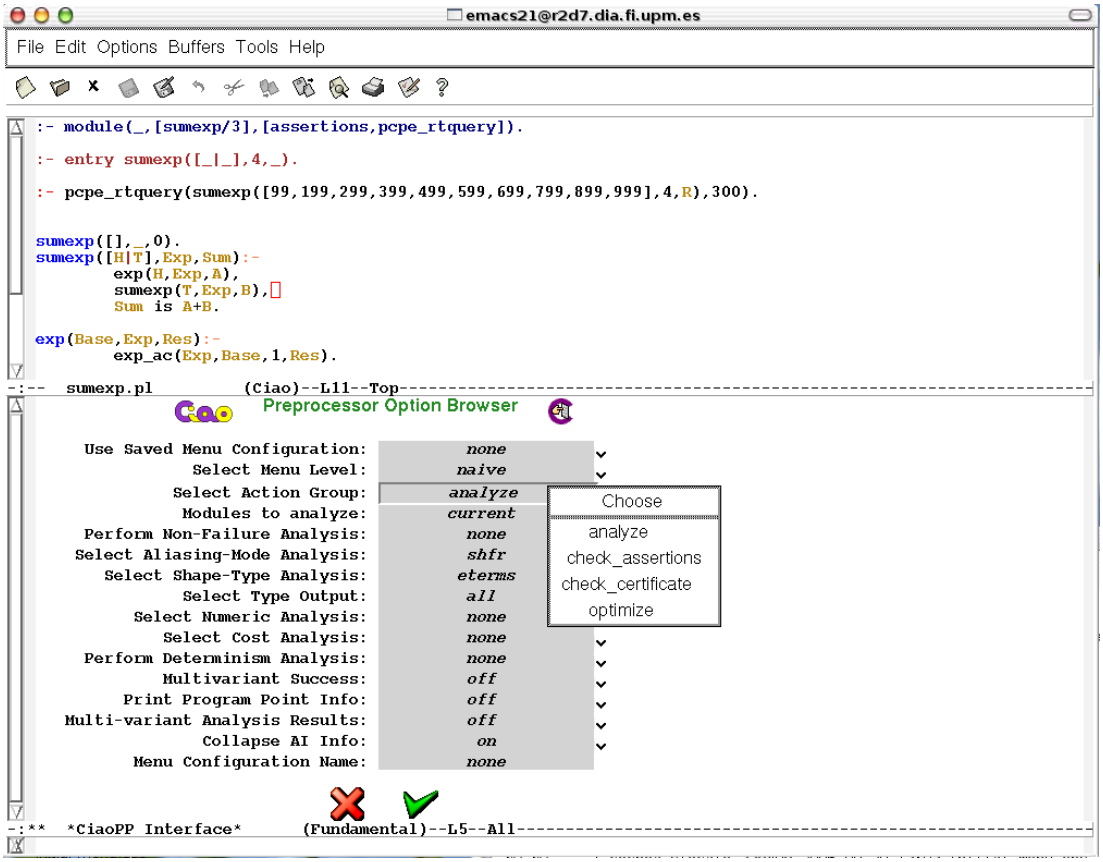

Figure 11.1: Starting Menu for Browsing `CiaoPP` Options.

- certificate checking (`check_certificate`) or
- program optimization (`optimize`).

The relevant options for the selected `action group` are then shown, together with the relevant flags.

In order to perform poly-controlled partial evaluation of the current `Ciao` program, `optimize` should be chosen as an `action group` from the initial menu.

`CiaoPP` provides several kinds of program optimizations, as shown in Figure 11.2:

- traditional partial evaluation (`spec`),
- parallelization (`parallelize`),

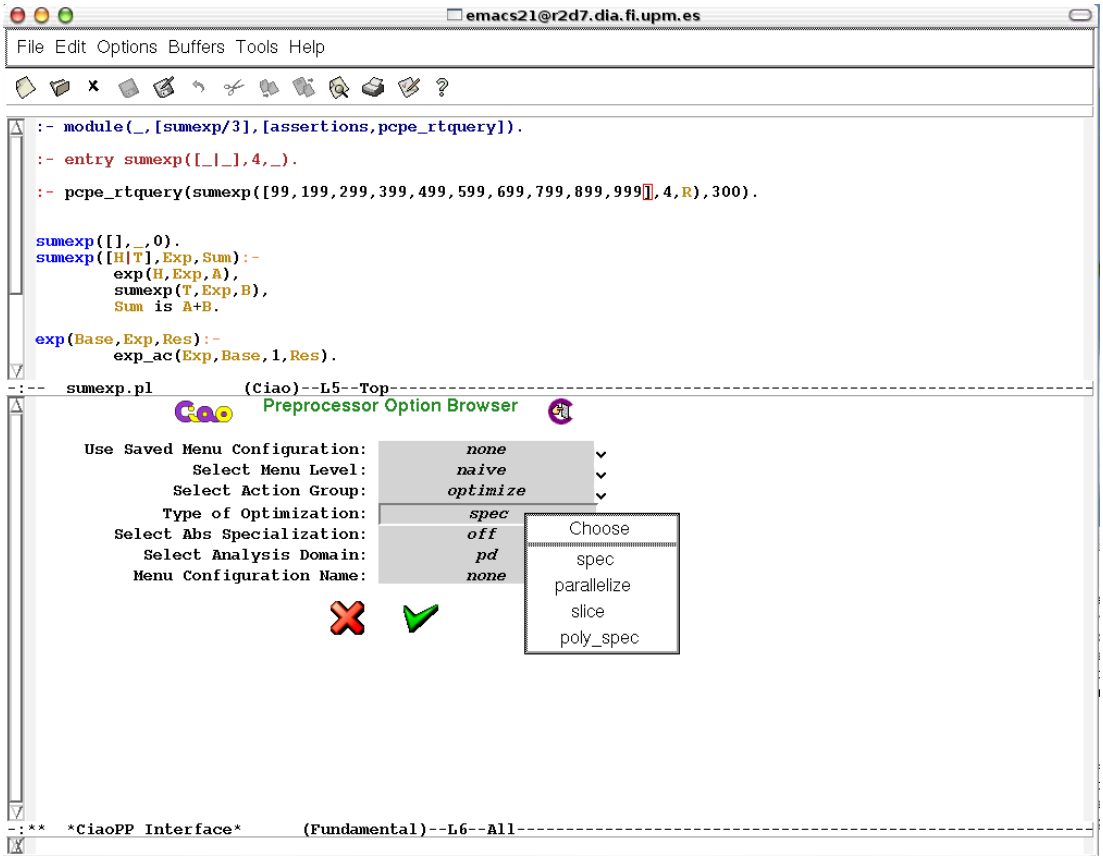

Figure 11.2: Optimization Menu.

- slicing (`slice`), and
- poly-controlled partial evaluation (`poly_spec`).

In the next section we explore the different options available in `CiaoPP` for performing poly-controlled partial evaluation.

11.3 Available Options for Poly-Controlled Partial Evaluation

`CiaoPP` provides two kinds of menu levels for users, a naïve one and an expert one.

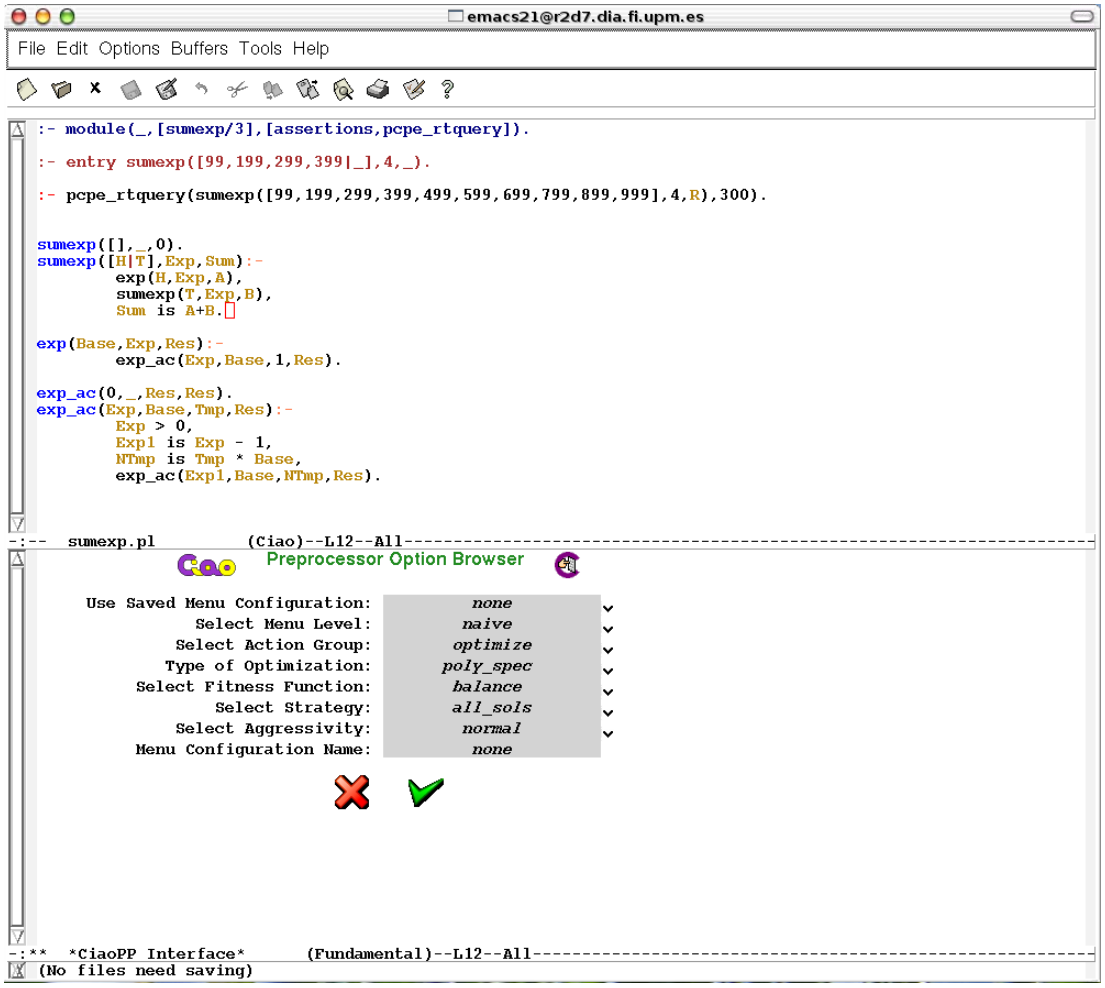

Figure 11.3: NaÏve Mode for Poly-Controlled Partial Evaluation.

11.3.1 Options for Naïve Users

As shown in Figure 11.3, in order to apply poly-controlled partial evaluation to a given program using the naïve menu, the user needs to specify only three options:

fitness function: it allows to specialize a program focusing on

speedup: time-efficiency.

memory: memory usage.

bytecode: disk usage.

balance: both time-efficiency and space-efficiency.

bounded_size: both time-efficiency and space-efficiency.

As described in Appendix A, when time-efficiency or memory usage is to be measured, then the user needs to also specify a non-empty set of runtime queries, which can be added in the source file by using the `pcpe_rtquery/1` directive provided in `CiaoPP` by the package of the same name.

strategy: it selects the algorithm of poly-controlled partial evaluation to be used:

all_sols: the original search-based PCPE algorithm (PCPE_{all}) presented in Chapter 5. By default, the predicate-consistency heuristic pruning is performed in this case, in order to try to ensure termination while still achieving good results.

oracle: the oracle-based PCPE algorithm presented in Chapter 10 (O-PCPE) is available through this option. As described in such chapter, this algorithm is very efficient, although finding the solution of maximal fitness is not guaranteed. However, it will find in general good solutions, and it is an excellent choice for resource aware program specialization.

aggressivity: it selects the set of control strategies to be used by the poly-controlled partial evaluator. This is one of the most user-friendly options available for naïve users, since the system automatically decides the combination of global and local control rules to be used based on the aggressivity level chosen by the user. Expert users can still play with any combination of control strategies (including newly added or custom ones) from the top level, as we will see in Section 11.4.

conservative: selects a set of conservative control rules, e.g. containing deterministic unfolding strategies. The main advantage is that the specialization process will terminate in general.

normal: is an intermediate level, with non-leftmost unfolding strategies but with a unfolding branching level set to 1.

aggressive: uses aggressive rules, e.g. non-leftmost unfolding with unlimited unfolding branching level. It is the most aggressive and sometimes it will get the best results, but depending on the strategy being used, it could run out of memory. It is better if used with some of the pruning techniques.

11.3.2 Options for Expert Users

Besides the options available for naïve users, poly-controlled partial evaluation can be further tweaked using some options available in expert mode, as shown in Figure 11.4.

In this mode, the user can control the behaviour of poly-controlled partial evaluation through the use of the following options:

Select Pruning: it allows to perform pruning as explained in Chapters 8 and 9. This option is only visible when *all_sols* has been selected as the PCPE algorithm to be used. The available pruning options are the following:

none: the PCPE_{all} algorithm from Chapter 5 is applied (no pruning).

heuristic: the heuristic-based PCPE algorithm (H-PCPE) from Chapter 8 is applied. The heuristic to be applied is selected through the *Heuristic* option (see below).

bnb: applies the branch and bound-based PCPE algorithm (BnB-PCPE) from Chapter 9. The depth at which pruning is performed is selected through the *Depth of Pruning* option (see below).

both: The HB-PCPE algorithm is applied when selecting this option.

Select Heuristic: it allows to perform heuristic pruning, and it is available only if *all_sols* is the chosen strategy, and either *heuristic* or *both* are the selected pruning techniques to be applied to the search-based PCPE algorithm. Among the possible options under this menu we can find:

pred: the *predicate-consistency* heuristic described in Chapter 8 is to be applied over the search space of the search-based poly-controlled partial evaluation algorithm.

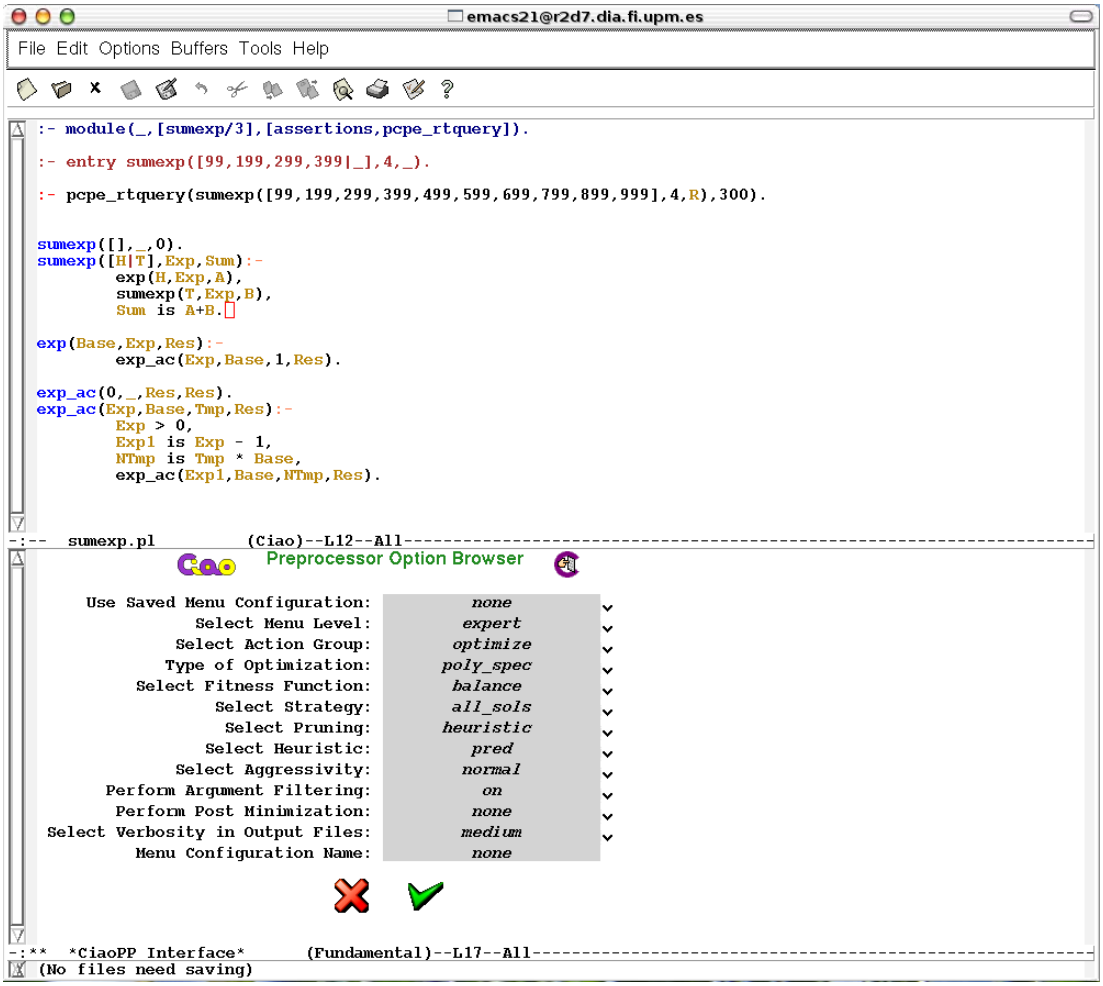

Figure 11.4: Expert Mode for Poly-Controlled Partial Evaluation.

modes: the *mode-consistency* heuristic described in Chapter 8 is to be applied over the search space of the search-based poly-controlled partial evaluation algorithm by selecting this option. In this case, the different domains of modes can be selected with the option `Select Modes Domain` (see below).

Select Modes Domain: this option is available only if the selected strategy is *all_sols*, either *heuristic* or *both* are the selected pruning techniques, and the heuristic is set to *modes*. In this case, the following domains (as described in Chapter 8) are available to the specializer:

sd: arguments of a call to a predicate are abstracted to one of two values in the domain $\{s, d\}$, where s stands for *static*, and d stands for *dynamic*.

sdl: the *sd* domain is extended by taking into account list skeletons.

sd_depth2: we use the *sd* domain, applied to a level of depth 2.

Select Depth Limit: this option is available only if the selected strategy is *all_sols* and either *bnb* or *both* are the selected pruning techniques to be applied to the search-based PCPE algorithm, and sets the depth in the tree at which pruning will take place. For instance, if set to 3, every 3 levels pruning will be performed. This value must be greater than 0. If this is not the case, then the specializer will set it to 3 by default.

Perform Argument Filtering: it determines whether redundant arguments in specialized predicates can be removed (see for example [41, 9, 44]). This can result in smaller programs.

Perform Post Minimization: it selects whether to perform a post minimization step, as described in Chapter 4. This can be especially useful when focusing on space-efficiency, in order to obtain smaller programs. The different options for this menu entry are described in the same chapter.

Select Verbosity in Output Files: poly-controlled partial evaluation can provide useful information (specially for debugging) in the form of comments on the output file. The amount of information provided is governed by this option:

none: no extra information is provided in the residual program.

medium: the percentage of each control strategy used to obtain all atoms in the global control is provided for each generated residual program. This allows to determine if such solution is obtainable by traditional partial evaluation.

high: in this mode, the following information is provided as comments in the output file:

- the percentage of each control strategy used to obtain all atoms in the global control.

- all atoms in the global control.
- a `.dot` file representing the explored PCPE-tree. This file can be inputted to the `dot` program [63] in order to generate a .ps representation of the PCPE-tree (or any other format supported by the `dot` program). This is useful to understand the behaviour of poly-controlled partial evaluation, and how pruning techniques work over this search space. Most PCPE-tree representations in this thesis have been generated with this option.
- a `.dat` file containing the fitness values of all candidate solutions found by poly-controlled partial evaluation. This file can be inputted to `gnuplot` [129] in order to have a representation of the distribution of fitness values. Examples of these representations generated by this framework have been used in Chapter 6.

After setting all flags, the user just need to click on the  icon and poly-controlled partial evaluation takes place. When PCPE finishes it will automatically load the best solution in the second buffer, as shown in Figure 11.5.

11.4 A Session Example for Expert Users

As we have mentioned before, existing partial evaluators usually offer a wide set of parameters to choose from, most of them affecting the quality of the residual program obtained.

In our implementation of the poly-controlled partial evaluator, besides providing a graphical menu with very few options, intended for the naïve user, and with some more extra options for the advanced user, we also provide the possibility of setting low-level flags modifying the behaviour of poly-controlled partial evaluation.

11.4.1 A PCPE Session in the Top Level of Ciao

In this section we show an example session for expert users, explaining the available flags and the effects on the achieved specialization that is controlled by these flags.

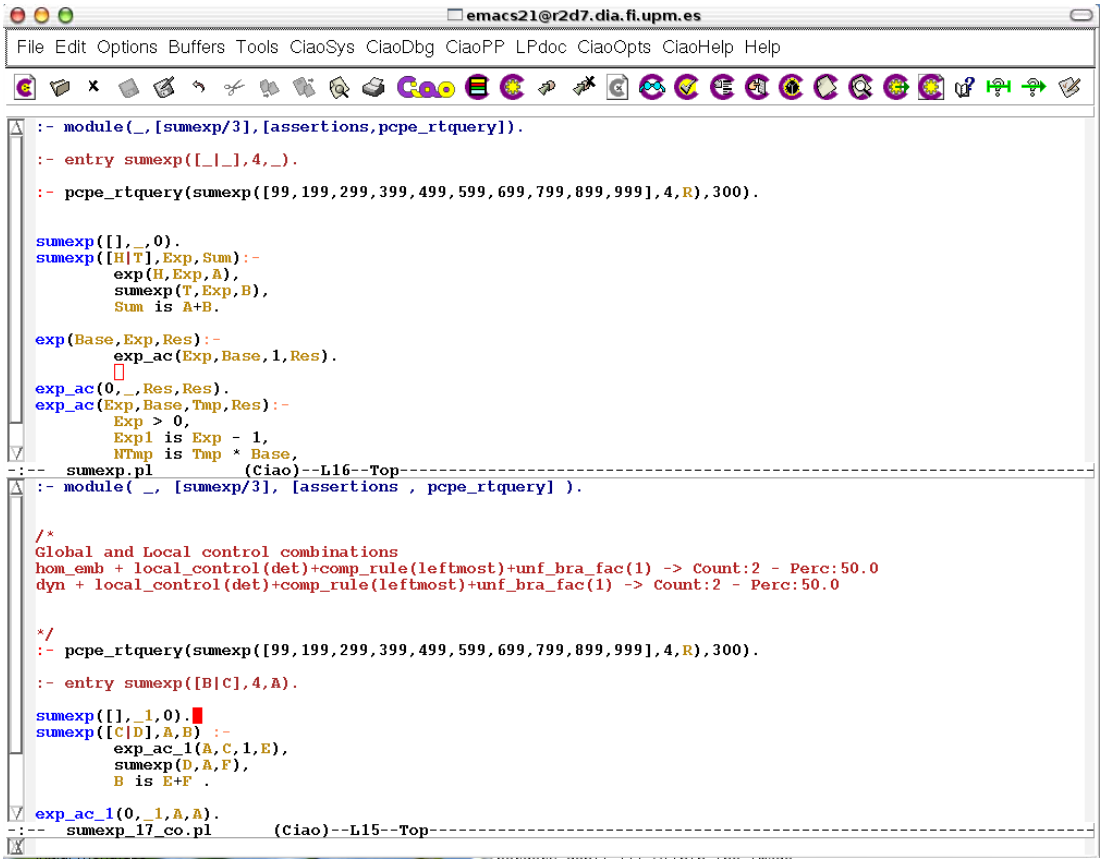

Figure 11.5: Residual Program Obtained by Poly-Controlled Partial Evaluation.

In order to start a CiaoPP session, the top level of Ciao should be started by typing ciao in a shell, or M-x ciao from Emacs. From the top level of Ciao, we can load CiaoPP by issuing:

```
:- use_module(ciaopp(ciaopp)).
```

Once we have loaded CiaoPP, we can set the corresponding flags by using the CiaoPP predicate set_pp_flag/1. We list the currently available flags below. The only mandatory flag that needs to be set is the fixpoint flag, which controls the fixpoint algorithm to be used during analysis. In order to use poly-controlled partial evaluation, you should set this flag to poly_spec:

```
:- set_pp_flag(fixpoint, poly_spec).
```

After setting the corresponding flags, we can load the module to be analyzed by using the CiaoPP predicate module/1:

```
:- module(foo).
```

where foo.pl is in the current directory. Otherwise we should use an absolute path. Now we can analyze this module using the CiaoPP predicate analyze/1.

```
:- analyze(pd).
```

After the analysis is completed, all solutions found by PCPE are asserted. Code can be generated for all of them, and the evaluation step takes place by calling the CiaoPP predicate transform/1.

```
:- transform(codegen_poly).
```

In the top-level a message will inform of which generated solution is the best one.

11.4.2 Available Flags for Controlling PCPE from the Top Level

The following flags are available from the top level, and they allow to drive the behaviour of poly-controlled partial evaluation:

poly_global_control Determines the set of abstraction functions to be used during poly-controlled global control. This flag is set to a list of global control rules, where the valid values are taken from the possible values of the flag global_control: hom_emb, hom_emb_num, dyn, id and inst (see [20] for further information on these values).

For example, in order to use {*hom_emb*, *dynamic*} as a set global control rules the following CiaoPP predicate call should be issued:

```
:- set_pp_flag(poly_global_control,[hom_emb, dyn]).
```

poly_local_control Determines the set of unfolding strategies to be used during poly-controlled global control. This flag is set to a list of local control rules, where a local control rule is a list containing the unfolding strategy (local_control), the computation rule (comp_rule) and the unfolding branching factor (unf_bra_fac) (the last two are optional).

local_control: is the actual unfolding strategy. It can take any of the following values: `orig, inst, det, det_la, depth, all_sol, hom_emb, hom_emb_anc, hom_emb_as, df_tree_hom_emb, df_hom_emb, df_hom_emb_as` (see [20] for further information on these values).

comp_rule: determines the computation rule to be used. It can take any of the following values: `leftmost,safe_jb,bind_ins_jb,no_sideff_jb, jump_builtin, eval_builtin, local_emb`. See [20] for further information on these values.

unf_bra_fac is a non-negative integer determining the unfolding branching factor to be used during unfolding.

For example, in order to set the local control rules to a set containing an aggressive and a conservative control rules, the following `CiaoPP` predicate call can be issued:

```
:- set_pp_flag(poly_lobal_control,
      [[local_control(det),
        comp_rule(leftmost),
        unf_bra_fac(1)],
       [local_control(df_hom_emb_as),
        comp_rule(bind_ins_jb),
        unf_bra_fac(0)]]).
```

poly_fitness: specifies the fitness function to be used by the poly-controlled partial evaluator. The possible values are `speedup, bytecode, memory, balance, bounded_size`, as described in Appendix A. The default value is `bytecode`.

pcpe_bounded_size: determines the maximum size of the residual program. It can be expressed in bytes, with a suffix (e.g. 5890, 10K, 2M), or as a factor of the size of the original program (e.g. 1.5x).

poly_strategy: determines the kind of PCPE algorithm to be used. The possible choices are

all_sols: corresponds to the search-based PCPE algorithm described in Chapter 5. The pruning to be performed on this algorithm is determined by the `poly_pruning` flag (see below).

oracle: corresponds to the oracle-based algorithm described in Chapter 10.

poly_pruning: determines the kind of pruning to be performed to the PCPE_{all} algorithm. The possible choices are

none: no pruning is performed.

heuristic: applies one of the heuristics explained in Chapter 8. The type of heuristic to be applied is set by the *polyvar_pcpe* flag.

bnb: applies the branch and bound pruning described in Chapter 9.

both: applies a combination of the branch and bound and the heuristics pruning techniques.

polyvar_pcpe: Controls the polyvariance of poly-controlled partial evaluation. The possible values are

off: no control of polyvariance is performed.

pred: the *predicate-consistent* pruning technique described in Chapter 8 is applied to the base algorithm.

modes: the *mode-consistent* pruning technique described in Chapter 8 is applied to the base algorithm. The domain used is set by the `poly_modes` flag, which takes one of the following values: `sd, sdl, sd_depth2`.

poly_depth_lim: is a non-negative integer value `N`. This number means that every `N` levels of depth in the search space tree, branch and bound pruning will be performed. If `N=0` then no pruning is performed.

output_info: this flag can be set to either `none, medium`, or `high`, and determines the amount of information to be written as comments in the residual files.

inter_opt_arg_filt: when set to `on`, redundant arguments in predicates are filtered away.

min_crit: determines whether a post-minimization step will be performed for all found solutions. It can take any of the following values: `none, equal, codemsg, nobindings, bindings, residual`, as described in Chapter 4.

All of these flags can be set by using the `CiaoPP` predicate `set_pp_flag/1`, in the following manner:

```
:- set_pp_flag(flag, value).
```

Chapter 12

Conclusions

The main motivation of this thesis has been to devise a *resource-aware* partial evaluation framework. Research and implementation of partial evaluation schemes have mainly focused on improving the time efficiency of programs, leaving factors such as size of the specialized program out of the picture.

Among the most relevant conclusions that can be drawn from this thesis we can mention:

- We have introduced the first partial evaluation framework that can produce specialized (hybrid) programs applying different control strategies to different call patterns (atoms). This scheme is called poly-controlled partial evaluation (PCPE). We have performed several experiments showing that, in many situations, these hybrid programs perform better (according to a given fitness function) than pure programs (as obtained by using traditional partial evaluation), especially when specialization is resource-aware. The main advantages offered by poly-controlled partial evaluation are:

  It is a resource-aware approach.

  Poly-controlled partial evaluation is one of very few approaches to resource-aware program specialization. Our framework uses fitness functions which consider factors such as memory consumption, disk usage, execution time of the specialized program, as well as a combination on some of these factors, in order to evaluate the quality of residual programs.

It obtains better solutions than traditional partial evaluation.

Experiments show that in many situations hybrid solutions obtained by poly-controlled partial evaluation are of higher quality than those obtained by traditional partial evaluation. The difference of quality is more evident when considering resource-aware fitness functions, since partial evaluation usually focuses on obtaining faster programs, without considering other factors such as the existing resources. Besides, since these solutions are hybrid, i.e., they use different control rules for different atoms, they cannot be obtained with traditional partial evaluation schemes.

It allows using any new control strategies.

Our framework has been designed with extensibility in mind, and thus, new control strategies resulting from further research can be added to the framework without effort, and combined with existing ones. In fact, in our implementation, PCPE and PE share a common library of (global and local) control rules, and thus, both approaches can immediately use any newly added rule.

It is integrated in `CiaoPP`.

The framework has been successfully integrated into `CiaoPP`. A graphical menu helps naïve users to specialize programs without having to specify a considerable number of parameters. Several decisions are taken automatically for the user. Experienced users can still tweak parameters, add new control rules, and try different combinations of control rules, in order to obtain the maximum benefit out of the PCPE framework.

- We have presented an algorithm for implementing PCPE which gathers all solutions that are obtainable by applying the different specialization strategies to the considered atoms, and then selects the best of all solutions by means of a fitness function. We call this algorithm the search-based PCPE (PCPE_{all}). Its main advantage is that it obtains solutions of maximal fitness. Its main drawback is that it suffers from an inherent explosion of its search space. In this thesis, we have presented several schemes for prun-

ing the search space of search-based the poly-controlled partial evaluation algorithm.

  - An efficient pruning scheme is based on heuristics, where, as we have seen in our experiments, the solution of maximal fitness is preserved most of the times. This pruning works by considering only those configurations that are consistent with previously processed ones and pruning away the rest. We have described and implemented several abstraction domains for consistency checking, which have different degrees of precision. The search-based PCPE using this heuristic is called H-PCPE.
  - We have presented a pruning scheme based on branch and bound (BnB), which, although is more complex to implement, it guarantees finding a solution of maximal fitness. Also, the particular implementation described in this work allows to quickly find solutions, setting in this way good quality upper bounds, and helping in performing a more effective pruning. The search-based PCPE using this heuristic is called BnB-PCPE.
  - Heuristic-based and branch and bound-based pruning techniques can be effectively combined, achieving the highest pruning in terms of size of the resulting search space, and where the obtained solution is of maximal fitness and consistent with the abstraction used. The search-based PCPE using this combination of heuristics is called PB-PCPE.

- We have presented also an Oracle-based PCPE algorithm (O-PCPE), which depends on an oracle that decides which specialization strategy is better suited for a given atom. This algorithm traverses only one PCPE-path, finding only one specialized program. Thus, it introduces a constant, rather than exponential, overhead factor to the complexity of standard PE. At least in our experiments, O-PCPE obtains specialized programs which are significantly better than those generated by standard PE, especially when considering resource-aware fitness functions. This approach also allows playing with a larger number of control strategies.

- As explained above, in this thesis several PCPE algorithms have been implemented: PCPE$_{all}$, H-PCPE, BnB-PCPE, PB-PCPE, and O-PCPE. We believe that PCPE$_{all}$ is too expensive to be used in practice with realistic programs. However, O-PCPE is a *replacement for traditional PE when doing resource-aware specialization*. As shown in Chapter 10, this approach obtains specialized programs of much better quality than traditional PE while having a similar cost, both in terms of memory consumed and specialization time. PB-PCPE is better than H-PCPE, and BnB-PCPE, and should be used only if *quality of the specialized program* is crucial.

- A minimization phase has been defined for abstract multiple specialization, traditional partial evaluation and poly-controlled partial evaluation. The main advantages provided by the approach described in this thesis are:

  It handles programs with *external predicates*.

  We have tackled in an accurate way the case in which programs contain *external predicates*, i.e., predicates whose code is not defined in the program being specialized, and thus it is not available to the specializer. This is the first work in which *any* external predicates is considered, even those having *impure* features. This is a rather important contribution, since most Prolog programs in the real world contain external predicates, i.e., techniques aimed at *pure* Prolog programs can deal only with toy problems. Our extension allows dealing with *any* Prolog program.

  It can collapse non-equivalent versions.

  We have proposed an additional generalization of the notion of equivalence which introduces the possibility of collapsing versions which are not *strictly* equivalent. This is achieved by residualizing certain computations for external predicates which would otherwise be performed at specialization time. This allows automatically trading time for space and we believe it may open the door to very interesting applications of partial evaluation and can be of interest in the context of embedded and pervasive systems, where computing resources and storage are often limited.

It considerably reduces the size of residual programs.

We have shown some experiments where a considerable reduction of specialized programs is achieved, both in terms of number of predicate versions and of bytecode size.

Appendix A

Fitness Functions

Poly-controlled partial evaluation is a resource-aware approach. It can generate several candidate program specializations, which are compared using a *fitness function* that assigns a numeric value to each of these candidate specializations, reflecting how good the corresponding program is.

The framework is parametric w.r.t. the fitness function so that the specialization can be performed with different aims in mind. Sometimes we may be interested in achieving code which is as time-efficient as possible, whereas in other cases space-efficiency can be a primary aim.

Given a residual program P_{spec} obtained by partially evaluating an input program P_{orig} with regard to some set of atoms $\mathcal{A}$, the PCPE framework assess P_{spec} by using one of the following different *resource-aware* fitness functions, some of them in the spirit of those in [27].

In all cases we assume that the fitness function returns values in the interval $[0, \infty)$.

A.1 Fitness Function Speedup

The fitness function Speedup compares programs based on their time-efficiency, measuring run-time speedup w.r.t. the original program. It is computed as

$$speedup(P_{spec}) \;=\; \frac{Time(P_{orig})}{Time(P_{spec})}$$

where $Time(P_{spec})$ is the execution time taken by the specialized program P_{spec} to run a set of run-time queries, and $Time(P_{orig})$ is the time taken by the original program P_{orig} to perform the same task.

In this case, the user needs to provide a set of run-time queries with which to time the execution of the program. Thus, such queries should be representative of the real executions of the program.

We have implemented a package in `CiaoPP` called `pcpe_rtquery`. This package provides a directive called `pcpe_rtquery/1`, which takes as an argument a run-time query, i.e., a call to a predicate defined in the given module with some partially instantiated arguments.

The user can specify several runtime queries using several directives, as in the example in Listing A.1.

A.2 Fitness Function Bytecode

The fitness function Bytecode compares programs based on their space-efficiency, measuring reduction of size of compiled bytecode w.r.t. the original program. It is computed as

$$bytecode(P_{spec}) \;=\; \frac{Size(P_{orig}) - Size(P_{empty})}{Size(P_{spec}) - Size(P_{empty})},$$

where $Size(P_{spec})$ is the size of the compiled bytecode of the specialized program P_{spec}, $Size(P_{orig})$ is the size of the compiled bytecode of the original program P_{orig}, and $Size(P_{empty})$ is the size of the compiled bytecode of an empty program.

A.3 Fitness Function Memory

The fitness function Memory compares programs based on their space-efficiency, measuring reduction of the space taken in memory w.r.t. the original program. It is computed as

$$memory(P_{spec}) \;=\; \frac{Size(P_{orig})}{Size(P_{spec})},$$

where $Size(P_{spec})$ is an estimation of the memory consumption of the specialized program P_{spec}, and $Size(P_{orig})$ is an estimation of the memory consumption of

the original program P_{orig}.

In order to provide an accurate estimation of the memory usage of the specialized program, the user has to provide a runtime query by means of a `pcpe_rtquery/1` directive, as in the case of SPEEDUP. Then, during evaluation, a snapshot of the current memory is measured

- before loading the specialized program, and
- after loading the specialized program and running the runtime query (or queries).

The difference between these two measurements gives an estimation of the amount of memory taken by the specialized program.

A.4 Fitness Function BOUNDED_SIZE

The fitness function BOUNDED_SIZE imposes a maximum size over the bytecode of the residual program. PCPE selects the fastest solution out of those residual programs compliant with this restriction. In our framework, the maximum size M_{ax} of the obtained solution can be specified

- as an absolute number, in bytes or
- as a factor of the size of the original program, that gets translated to an absolute number of bytes.

The BOUNDED_SIZE fitness function is defined as

$$bounded\_size(P_{spec}, M_{ax}) = \begin{cases} 0 & \text{if } Size(P_{spec}) > M_{ax} \\ speedup(P_{spec}) & \text{otherwise} \end{cases}$$

where $Size(P_{spec})$ is the size of the compiled bytecode of the specialized program P_{spec}. If this size is greater than the maximum size allowed, then we assign 0 as fitness value of the specialized program, otherwise we assign to it the value resulting from applying the SPEEDUP fitness function to such a program.

Note that this fitness function is quite useful in pervasive computing. For instance, you could have a device with a hard constraint on the size of the program

to be loaded. This fitness function allows to select the fastest specialized program being compliant with such a restriction. Note also that is trivial to extend this function to measure the maximum memory that can be taken by the residual program, instead of its bytecode size.

A.5 Fitness Function Balance

The fitness function Balance is a combination of the Speedup and Bytecode fitness functions. It is defined as

$$balance(P) \;=\; speedup(P) \times bytecode(P),$$

and thus it takes into account both the size and the efficiency of the candidate programs. As it stands, it gives equal importance to both factors. It is easy to obtain variations of this formula which assign different weights to them, as best suited to each situation.

Listing A.1: Specifying Runtime Queries in the `rev/2` Example

```
:- module(_,[rev/2],[assertions, pcpe_rtquery]).
:- entry rev([_,_|L],R).

:- pcpe_rtquery(rev[1,2,3,6]).
:- pcpe_rtquery(rev[a,b,c|T]).

rev([],[]).
rev([H|L],R):-
        rev(L,Tmp),
        app(Tmp,[H],R).

app([],L,L).
app([X|Xs],Y,[X|Zs]):-
        app(Xs,Y,Zs).
```

Example A.5.1. *In this small example shown in Listing A.1, we can see the naïve reverse program, which is being specialized for lists containing at least two*

elements, as determined by the `entry/1` *directive contained in the* `assertions` *package. If we are running poly-controlled partial evaluation for specializing this program focusing on time-efficiency, or if memory consumption is our more valuable resource, then we need to provide one or more runtime queries using the* `pcpe_rtquery/1` *directive defined in* ***CiaoPP*** *in the package of the same name. In this particular example, we provide two representative list for testing the speed of specialized programs obtained by PCPE. The first call uses a closed list of 4 elements, while the second call uses an open list of a least 3 elements.*

Appendix B

Benchmark Programs

Many of the benchmark programs that have been used throughout the thesis to test the poly-controlled partial evaluation framework have been borrowed from Michael Leuschel's *Dozen of Problems of Partial Deduction* (DPPD) library [79], since they cover a wide range of different application areas, including pattern matching, databases, expert systems, meta-interpreters, etc. The DPPD library includes some benchmarks adapted from Lam and Kusalik's set of problems, which first appeared in [69]. The rest of benchmarks are taken from different sources, such as [84, 80], the internet, and different Prolog libraries.

All benchmarks have been adapted to use the `CiaoPP` assertion language [107] in order to provide precise descriptions of the initial call patterns, and also in order to provide the runtime queries for those fitness functions which require to make runs of the specialized programs.

The size of the compiled code in `Ciao` is given in parentheses.

advisor (7595 bytes) A very simple expert system, containing no builtins nor negations, and which can be fully unfolded, by Horváth [58].

analysis (39985 bytes) A semantic analyzer for simple Spanish sentences. When using traditional partial evaluation with aggressive control rules, the generated program can be quite big for the given specialization queries.

applast (4803 bytes) This is a benchmark by Michael Leuschel which contains no builtins nor negations, and that appends an element at the end of a list.

This benchmarks has been used to show the benefits of conjunctive partial deduction [30, 72]. More details can be found in [75].

boyer (36619 bytes) A Boyer-Moore theorem prover written by Evan Tick after the Lisp version by R. P. Gabriel. It contains several extra-logical features, such as cuts.

browse (12579 bytes) This is a program from the Gabriel benchmarks that browses a database, by Tep Dobry and Herve Touati. It includes several builtins, cuts, etc.

contains (5549 bytes) This Lam & Kusalik benchmark [69] is a highly non-deterministic and inefficient pattern matcher. This benchmark program uses the `(\==)/2` builtin.

credit (16932 bytes) A credit evaluation system, by Pedro Lopez García. The main predicate answers a request by a given client for a credit. It includes several builtins and calls to external predicates.

datetime (17689 bytes) The datetime benchmark implements a library containing predicates that perform logical arithmetic on dates and times. The distinction is most noticeable when dealing with months, which have varying numbers of days. The arithmetic is *pure* date arithmetic, in the sense that is it adds calendar months, so Feb 15th plus one month yields Mar 15th. Adding years over leap years winds up on same days as well. Dates are correctly fixed for the corner cases, so an intermediate result of Feb 30th will become Mar 2nd in a non leap year and Mar 1st in leap year.

depth (5702 bytes) A simple non-ground meta-interpreter which keeps track of the maximum length of refutations, by Lam & Kusalik [69]. It has to be specialized for a simple, fully unfoldable object program. It uses neither negations nor builtins.

doubleapp (4516 bytes) Naive implementation for a predicate that appends three lists, written using two calls to the ordinary `append/3` predicate. This program is inefficient because an intermediate variable is constructed by the first call to append and then traversed again by the second call to

append. It also tests whether deforestation [124] can be done. For further details, see Chapters 10 and 11 of [80].

example_pcpe (5504 bytes) Example used to show the benefits and the need of poly-controlled partial evaluation [110]. The code is shown in Listing 5.9 at Chapter 5. It contains builtins and cannot be fully unfolded.

ex_depth (6107 bytes) A (more difficult) variant of `depth`, using a different simple non-ground metainterpreter keeping track of the maximum length of refutations, with an object program which cannot be fully unfolded. It uses neither negations nor builtins.

exponential_peano (5639 bytes) A program calculating the exponential of a given number using Peano's arithmetic.

flip (4614 bytes) Simple deforestation example by Wadler [123] in which a tree structure is flipped twice (thus returning back to the original tree), and whose goal is to obtain a program which just copies the tree.

flattrees (4721 bytes) This benchmark takes a list of arithmetic expression trees, and flattens each expression tree to a list containing only the operands or the original expression. This benchmark contains the builtin for term construction or decomposition `(=..)/2`.

freeoff (4994 bytes) This benchmark implements a predicate checking that a given expression does not occur anywhere in another expression. If the second expression contains an unbound variable, the predicate must fail, since the first expression might occur there.

grammar (11381 bytes) A Lam & Kusalik [69] benchmark implementing a *DCG (Definite Clause Grammar)* parser which has to be specialized for a particular grammar. When transformed into ordinary clauses the builtin `=/2` appears.

groundunify_simple (10368 bytes) A ground unification algorithm calculating explicit substitutions which uses builtins and negation. Adapted from [31]. More details can be found in [73].

liftsolve_app (7111 bytes) A meta-interpreter for the ground representation (adapted from a "non-executable" but specializable one by John Gallagher [40], similar to the InstanceDemo by Hill and Gallagher [56]) which lifts the program to the non-ground representation for resolution. The goal is to specialize this meta-interpreter for `append` as the object program. Some details about this meta-interpreter can also be found in [82].

match (4781 bytes) A semi-naïve pattern matcher by Lam & Kusalik [69], whose goal is to obtain a Knuth-Morris-Pratt pattern matcher by specialization for the pattern "aab". This benchmark program uses the `(\==)/2` builtin.

match_append (4538 bytes) A very naïve string matcher, written with 2 appends. This benchmark contains no builtin's nor negations. A similar matcher has been used in [103, 102].

mmatrix (5261 bytes) This program implements the multiplication of matrices. It uses several arithmetic builtins.

nrev (4623 bytes) The naïve reverse algorithm used in Chapter 7 to illustrate the explosion of the search space of poly-controlled partial evaluation. It does not contain builtins nor negations. With the specialization query used, this benchmark cannot be fully unfolded.

permute (4687 bytes) A program which computes all possible permutations of the elements of the input list. An important feature of this program is that its results, when fully unfolded, are much larger than the original program. The specialization query used is a fully-instantiated, closed list, and thus it can be fully unfolded.

petri_meta (5625 bytes) A metainterpreter for Petri nets with the net of the `petri-object` benchmark at the object level. The goal is to prove that for the Petri net at hand (and for any number of processes) there is no trace that leads to an unsafe state with more than 1 process in its critical section.

petri_object (4787 bytes) A reified version of the `petri-meta` benchmark. The goal is to prove that for the Petri net at hand (and for any number of processes) there is no trace that leads to an unsafe state with more than 1 process in its critical section.

prolog_read (28300 bytes) The original prolog parser by D.H.D. Warren and Richard O'Keefe. It reads Prolog terms in Dec-10 syntax. Modified by Alan Mycroft to regularise the functor modes, to make it both easier to understand, and also to fix some bugs concerning the curious interaction of cut with the state of parameter instantiation.

qplan (37512 bytes) Designed by D.H.D. Warren. It supplies the control information (i.e., sequencing and cuts) needed for efficient execution of a query.

qsort (5170 bytes) The classical quicksort algorithm, which includes arithmetic builtins. The specialization query used is an open list with a few elements instantiated, thus it cannot be fully unfolded.

qsortapp (5390 bytes) A naïve quicksort algorithm implemented using `append`. It contains arithmetic builtins. With the specialization query used it cannot be fully unfolded.

relative (5909 bytes) A Lam & Kusalik [69] benchmark consisting of a family database. It contains neither builtins nor negations. With the specialization query considered it can be fully unfolded.

remove (4929 bytes) Sophisticated deforestation example, by Jesper Jorgensen. This benchmark program uses the `(\==)/2` builtin, and it cannot be fully unfolded.

rev_acc_type (4575 bytes) This benchmark is difficult in the sense that it causes the generation of an infinite number of characteristic trees in a quite natural manner. Indeed, the program is simply the well-known reverse with accumulating parameter program to which a type check on the accumulator has been added. In that way the growth of the accumulator causes a growth of the type checking computation, and thus a growth of the characteristic tree describing that computation. Further details can be found in [74].

rotateprune (5692 bytes) A more sophisticated deforestation example by [105]. The program rotates and prunes a binary tree, and the goal is to deforest the intermediate tree used between the two operations. This benchmark contains no builtins nor negations. Further details can be found in [46].

ssuply (9213 bytes) Another simple expert system, with simple builtins, by [69]. It can be fully unfolded.

sublists (5638 bytes) A predicate taking a list of pairs of numbers as the first argument, and an arbitrary list as a second argument. Every pair of numbers of the first list denotes the beginning and end of a sublist of the second argument. Sublists are returned in the third argument of the predicate. This benchmark contains builtins. For the specialization query considered it cannot be fully unfolded.

sumexp (4918 bytes) A program which applies the exponential function to all elements of a given list, and then returns the sum of all new elements in the list. It cannot be fully unfolded, and it makes use of arithmetic builtins.

transpose (5005 bytes) A program transposing matrices of any dimension, using uses neither negation nor builtins, by [69]. Also in [40].

vanilla_db (13446 bytes) A vanilla meta-interpreter, with a contrived object program invented by Bart Demoen, and borrowed from [27].

Appendix C

Program Slicing in CiaoPP

As mentioned in Chapter 1, another resource-aware program specialization technique is *program slicing*. Program slicing, originally introduced by Weiser [127, 128] in the context of imperative programming, is a general method for extracting the program sentences that potentially affect (or are affected by) some criterion (e.g., a program point, a variable, a procedure, etc), usually referred to as a *slicing criterion*. Program slices are often computed from a *program dependence graph* [39, 68] that makes explicit both the data and control dependences for each operation in a program. Program dependences can be traversed backwards and forwards—from the slicing criterion—giving rise to so-called *backward* and *forward* slices, respectively. Additionally, slices can be *dynamic* or *static*, depending on whether a concrete program's input is provided or not.

Program slicing was first proposed as a debugging technique to allow a better understanding of the portion of code which revealed an error. Since then, it has been successfully applied to a wide variety of software engineering tasks, like program understanding, debugging, testing, differencing, specialization, merging, etc. More detailed information on slicing for imperative programs can be found in the surveys of Harman and Hierons [53] and Tip [120]. Although it is not so popular in the declarative programming community, several slicing techniques for declarative programs have also been developed during the last decade (see, e.g., [47, 76, 77, 96, 111, 116, 119, 122, 99, 96]).

C.1 Program Slicing for Specializing Logic Programs

In [111], a specialization method for *strict* first-order functional programs based on *static* slicing was presented. Basically, given a program P and a projection function π, [111] extracts a program that behaves like $\pi(P)$ (roughly, by symbolically pushing π backwards through the body of P).

Then, [122] introduced a novel approach to forward slicing of *lazy* functional logic programs, exploiting the similarities between slicing and partial evaluation to compute forward slices by a slight modification of an existing partial evaluation scheme [3]. This work was adapted to the logic programming paradigm and extended in [77].

In [77], a *slicing criterion* is simply a goal. A forward slice then contains a *subset* of the original program with those clauses that are reachable from the slicing criterion. Similar to [116], this notion of "subset" is formalized in terms of an abstraction relation, in order to allow arguments to be removed or replaced by a special term. The algorithm proposed in [77] relies on applying partial evaluation to a given program P and a goal G, and using characteristic trees (described in Chapter 4) to record the clauses used in every unfolding operation performed. Finally, in the code generation phase, all used clauses are collected from the characteristic trees, and their original definitions are used to build the slice. In general, slices will contain redundant arguments that are not relevant for the execution of the slicing criterion. These computed slices can be further refined by redundant argument filtering transformations [101].

Based on these ideas, we have implemented a slicer in the program development system `Ciao` [17], and integrated it into `CiaoPP` [20, 55, 19], the preprocessor of `Ciao`.

C.2 A Slicing Session in CiaoPP

As seen in Chapter 11, a `CiaoPP` session consists in the preprocessing of a file. The session is governed by a menu, where the user can choose the kind of preprocessing to be done to a file from among several analyses and program transformations available.

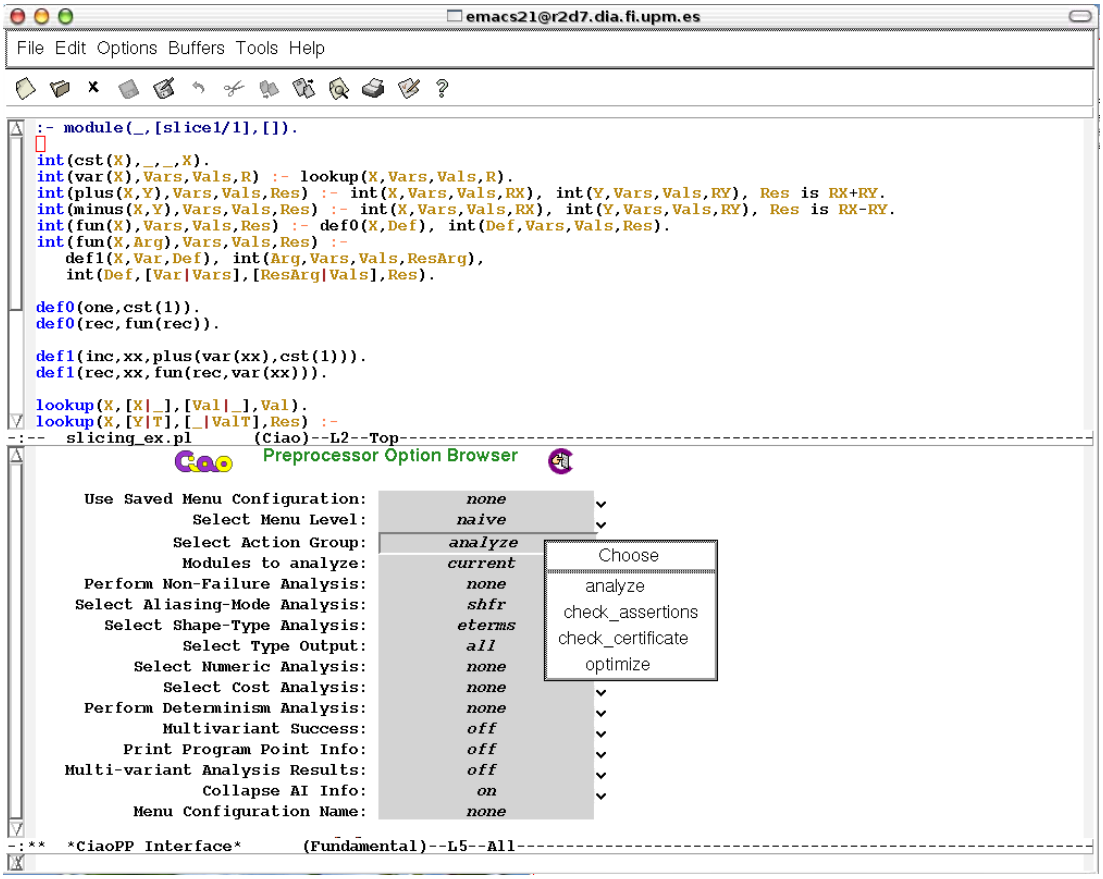

Figure C.1: Starting Menu for Browsing `CiaoPP` Options.

Clicking on the  icon in the buffer containing the file to be preprocessed, displays the initial menu, which will look (depending on the options available in the current `CiaoPP` version) something like the "Preprocessor Option Browser" shown in Figure C.1.

Except for the first and last lines, which refer to loading or saving a menu configuration (a predetermined set of selected values for the different menu options), each line corresponds to an option the user can select, each having several possible values.

The preprocessing available in `CiaoPP` is located under the `Select Action Group` dropdown menu. The user can select either

- analysis (`analyze`) or

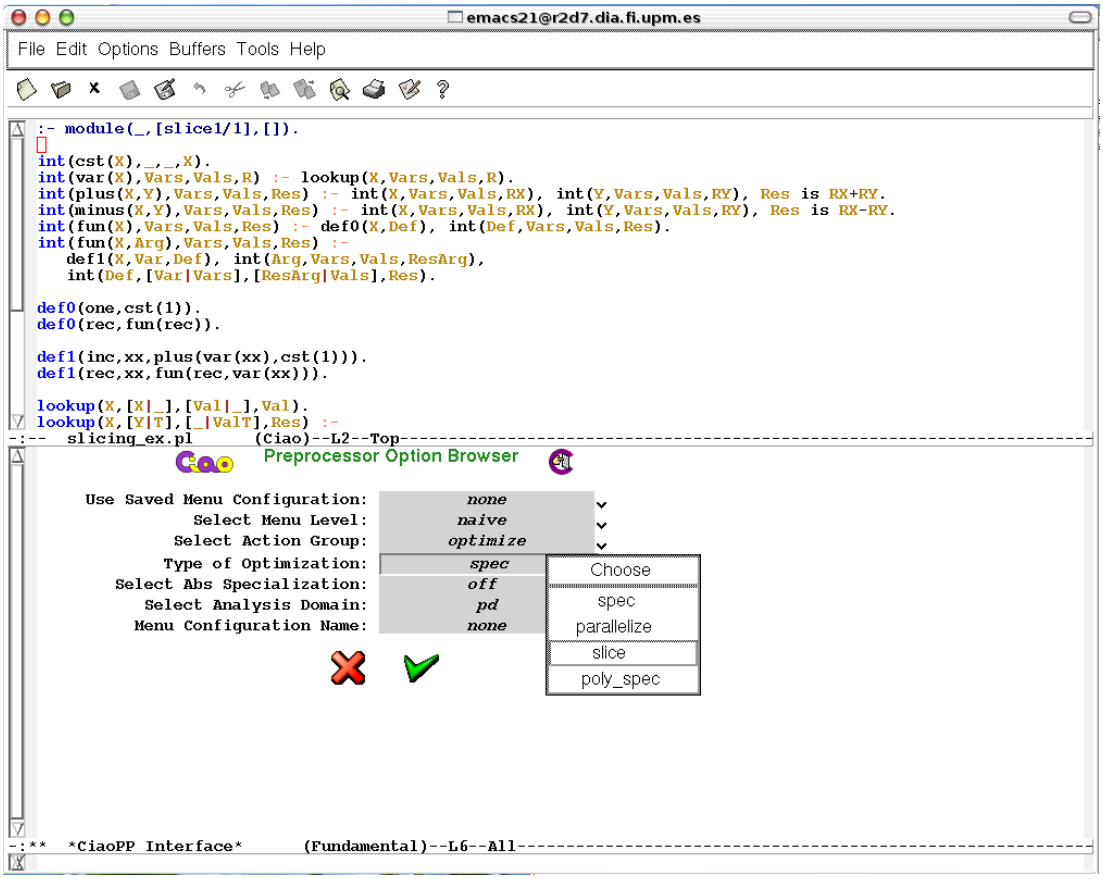

Figure C.2: Optimization Menu.

- assertion checking (`check_assertions`) or
- certificate checking (`check_certificate`) or
- program optimization (`optimize`).

The relevant options for the selected `action group` are then shown, together with the relevant flags.

In order to obtain a slice of the current `Ciao` program, `optimize` should be chosen as an `action group` from the initial menu.

`CiaoPP` provides several kinds of program optimizations, as shown in Figure C.2:

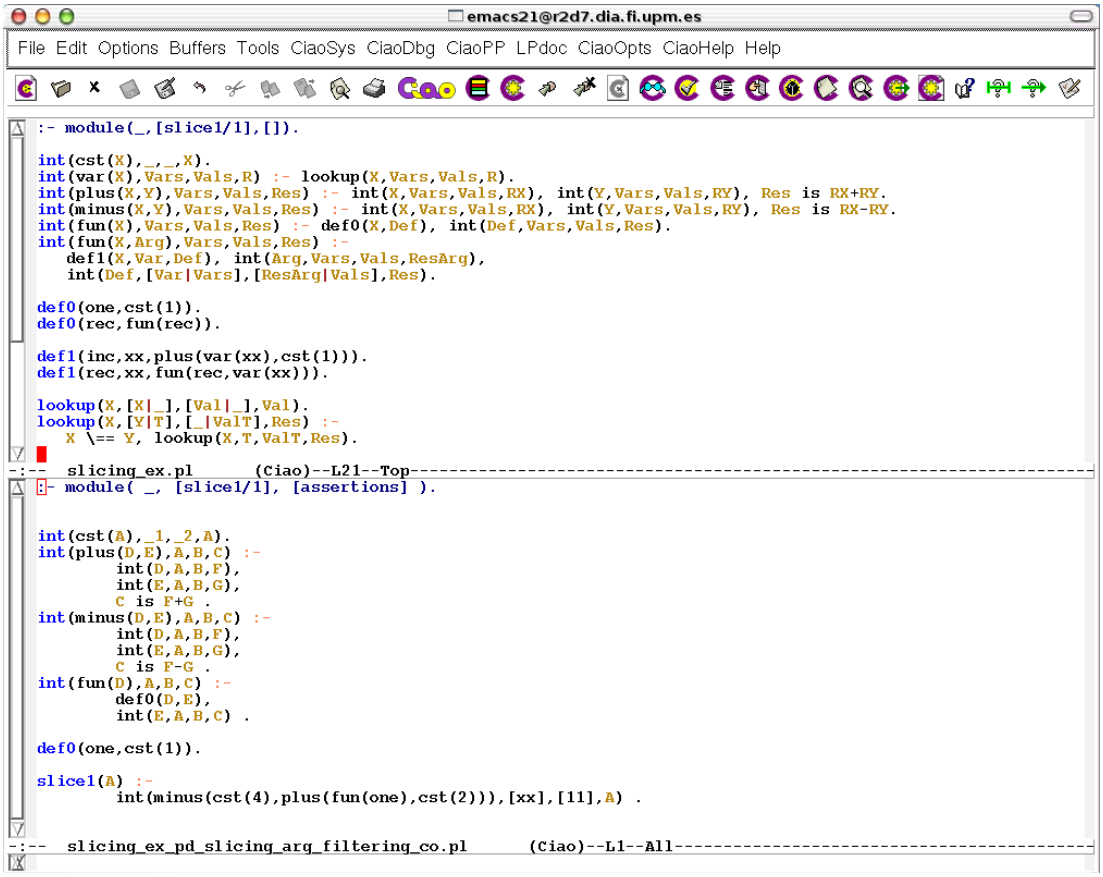

Figure C.3: Slice of the Original Program

- traditional partial evaluation (`spec`),
- parallelization (`parallelize`),
- slicing (`slice`), and
- poly-controlled partial evaluation (`poly_spec`).

By selecting `slice` in this menu, and clicking on the ✔ icon we will obtain the corresponding slice, which will be automatically loaded in the second buffer, as shown in Figure C.3.

In `CiaoPP`, and similarly to the case of partial evaluation described in Chapter 7, the description of initial queries (i.e., the slicing criterion) is obtained by

taking into account the set of predicates exported by the module, in this case {`slice/1`}, possibly qualified by means of *entry* declarations.

Bibliography

[1] E. Albert, M. Hanus, and G. Vidal. A Practical Partial Evaluation Scheme for Multi-Paradigm Declarative Languages. *Journal of Functional and Logic Programming*, 2002(1), 2002.

[2] E. Albert, G. Puebla, and J. Gallagher. Non-Leftmost Unfolding in Partial Evaluation of Logic Programs with Impure Predicates. In *Proceedings of the 15th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, number 3901 in *Lecture Notes in Computer Science*. Springer-Verlag, April 2006.

[3] E. Albert and G. Vidal. The Narrowing-Driven Approach to Functional Logic Program Specialization. *New Generation Computing*, 20(1):3–26, 2002.

[4] M. Alpuente, M. Falaschi, and G. Vidal. Narrowing-driven Partial Evaluation of Functional Logic Programs. In H.R. Nielson, editor, *Proceedings of the European Symposium on Programming (ESOP'96)*, volume 1058 of *Lecture Notes in Computer Science*, pages 45–61. Springer-Verlag, Berlin, 1996.

[5] Lars Ole Andersen. Partial Evaluation of C and Automatic Compiler Generation. In U. Kastens and P. Pfahler, editors, *Proceedings of Compiler Construction. 4th International Conference*, volume 641 of *Lecture Notes in Computer Science*, pages 251–257, Paderborn, Germany, 1992. Springer-Verlag.

[6] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[7] K.R. Apt. Introduction to Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Model and Semantics, pages 495–574. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.

[8] K.R. Apt and R. Bol. Logic Programming and Negation: A Survey. *Journal of Logic Programming*, 19&20, 1994.

[9] K. Benkerimi and P.M. Hill. Supporting Transformation for the Partial Evaluation of Logic Programs. In *Journal of Logic and Computation*, volume 3(5), pages 469–486, October 1993.

[10] Andrew Berlin and Daniel Weise. Compiling Scientific Code Using Partial Evaluation. *IEEE Computer*, 23(12):25–37, 1990.

[11] R. Bol. Loop Checking in Partial Deduction. *Journal of Logic Programming*, 16(1&2):25–46, 1993.

[12] Anders Bondorf. A Self-Applicable Partial Evaluator for Term Rewriting Systems. In J. Díaz and F. Orejas, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT'89)*, volume 352 of *Lecture Notes in Computer Science*, pages 81–95, Barcelona, Spain, 1989. Springer-Verlag.

[13] Anders Bondorf. Automatic Autoprojection of Higher Order Recursive Equations. *Science of Computer Programming*, 17(1-3):3–34, 1991.

[14] Antony F. Bowers and Corin A. Gurr. Towards Fast and Declarative Meta-Programming. In K. R. Apt and F. Turini, editors, *Meta-logics and Logic Programming*, pages 137–166. MIT Press, 1995.

[15] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.

[16] M. Bruynooghe, D. De Schreye, and B. Martens. A General Criterion for Avoiding Infinite Unfolding during Partial Deduction. *New Generation Computing*, 1(11):47–79, 1992.

[17] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. The Ciao Prolog System. Reference Manual. The Ciao System Documentation Series–TR CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM), August 1997. System and on-line version of the manual available at `http://clip.dia.fi.upm.es/Software/Ciao/`.

[18] F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Global Analysis of Standard Prolog Programs. In *Proceedings of the European Symposium on Programming (ESOP'96)*, number 1058 in *Lecture Notes in Computer Science*, pages 108–124, Sweden, April 1996. Springer-Verlag.

[19] F. Bueno, P. López-García, G. Puebla, and M. Hermenegildo. A Tutorial on Program Development and Optimization using the Ciao Preprocessor. Technical Report CLIP2/06, School of Computer Science, Technical University of Madrid (UPM), 28660 Boadilla del Monte, Madrid, Spain, January 2006.

[20] F. Bueno, P. López-García, G. Puebla, and M. Hermenegildo. The Ciao Prolog Preprocessor. Technical Report CLIP1/06, School of Computer Science, Technical University of Madrid (UPM), 28660 Boadilla del Monte, Madrid, Spain, January 2006.

[21] B. Le Charlier, S. Rossi, and P. Van Hentenryck. Sequence Based Abstract Interpretation of Prolog. *Theory and Practice of Logic Programming*, 2(1):25–84, 2002.

[22] B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems*, 16(1):35–101, 1994.

[23] D. Cohen, M. Cooper, P. Jeavons, and A. Krokhin. Identifying Efficiently Solvable Cases of Max CSP. *Proceedings of the 21st Symposium on Theoretical Aspects of Computer Science (STACS'04)* pages 152–163, Le Corum, Montpellier, France, 2004.

[24] A. Colmerauer. Opening the Prolog-III Universe. In *BYTE Magazine*, August 1987.

[25] Charles Consel. A Tour of Schism: a Partial Evaluation System for Higher-Order Applicative Languages. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'93)*, pages 145–154, Copenhagen, Denmark, 1993. ACM Press.

[26] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[27] Stephen-John Craig and Michael Leuschel. Self-Tuning Resource Aware Specialisation for Prolog. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'05)*, pages 23–34, New York, NY, USA, 2005. ACM Press.

[28] O. Danvy, N. Hentze, and K. Malmkjær. Resource-Bounded Partial Evaluation. *ACM Computing Surveys*, 28(2):244–247, June 1996.

[29] M. García de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 18(5):564–615, September 1996.

[30] D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M.H. Sørensen. Conjunctive Partial Deduction: Foundations, Control, Algorihtms, and Experiments. *Journal of Logic Programming*, 41(2&3):231–277, 1999.

[31] D. A. de Waal and J.P. Gallagher. Specialisation of a Unification Algorithm. In T.P. Clement and K.-K. Lau, editors, *Proceedings of the 1st International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'91)*, pages 205–220, Manchester, UK, 1991.

[32] Saumya K. Debray. Profiling prolog programs. *Software Practice and Experience*, 18(9):821–839, 1983.

[33] S. K. Debray and N. W. Lin. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.

[34] Saumya K. Debray. Resource-Bounded Partial Evaluation. In *Proceedings of the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'97)*, pages 179–192. Amsterdam, The Netherlands, 1997. ACM Press.

[35] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard*. Springer-Verlag, 1996.

[36] N. Dershowitz and J.P. Jouannaud. *Rewrite Systems* In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 243–320. Elsevier, MIT Press, 1990.

[37] N. Dershowitz and J.P. Jouannaud. *Rewrite Systems* In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 243–320. Elsevier, MIT Press, 1990.

[38] Andrei P. Ershov. Mixed Computation: Potential applications and Problems for Study. *Theoretical Computer Science*, 18:41–67, 1982.

[39] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[40] J. Gallagher. A System for Specialising Logic Programs. Technical Report TR-91-32, University of Bristol, November 1991.

[41] J. Gallagher and M. Bruynooghe. Some Low-Level Transformations for Logic Programs. In M. Bruynooghe, editor, *Meta90 Workshop on Meta Programming in Logic*, pages 229–244, 1990.

[42] J. Gallagher and M. Bruynooghe. The Derivation of an Algorithm for Program Specialisation. *New Generation Computing*, 9(1991):305–333, 1991.

[43] J. Gallagher and L. Lafave. Regular Approximation of Computation Paths in Logic and Functional Languages. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation*, volume 1110, pages 115 – 136. Springer Verlag *Lecture Notes in Computer Science*, 1996.

[44] J.P. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proceedings of the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'93)*, pages 88–98.Copenhagen, Denmark, 1993. ACM Press.

[45] J.P. Gallagher and D.A. de Waal. Fast and Precise Regular Approximations of Logic Programs. In Pascal Van Hentenryck, editor, *Proceedings of the 11th International Conference on Logic Programming (ICLP'94)*, pages 599–613. MIT Press, 1994.

[46] R. Glück, J. Jorgensen, B. Martens, and M.H. Sorensen. Controlling Conjunctive Partial Deduction of Definite Logic Programs. Technical Report CW 226, Departement Computerwetenschappen, K.U. Leuven, Belgium, May 1996.

[47] V. Gouranton. Deriving Analysers by Folding/Unfolding of Natural Semantics and a Case Study: Slicing. In *Proceedings of the International Static Analysis Symposium (SAS'98)*, pages 115–133, 1998.

[48] C. A. Gurr. *A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel*. PhD thesis, Department of Computer Science, University of Bristol, January 1994.

[49] C. A. Gurr. Specialising the Ground Representation in the Logic Programming Language Gödel. In Y. Deville, editor, Logic Program Synthesis and Transformation. *Proceedings of the 3rd International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'93)*, Workshops in Computing, pages 124–140, Louvain-La-Neuve, Belgium, 1994. Springer-Verlag.

[50] Michael Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[51] Michael Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 80–93. ACM, 1997.

[52] Michael Hanus. Curry: An Integrated Functional Logic Language. Available at:
`http://www.informatik.uni-kiel.de/~curry/`, 2000.

[53] M. Harman and R.M. Hierons. An Overview of Program Slicing. *Software Focus*, 2(3):85–92, 2001.

[54] M. Hermenegildo, F. Bueno, G. Puebla, and P. López-García. Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In *Proceedings of the International Conference on Logic Programming (ICLP'99)*, pages 52–66, Cambridge, MA, November 1999. MIT Press.

[55] Manuel V. Hermenegildo, Germán Puebla, Francisco Bueno, and Pedro López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.

[56] Patricia Hill and John Gallagher. Meta-Programming in Logic Programming. In *Handbook of Logic in Artificial Intelligence and Logic Programming, Vol. 5.* Oxford Science Publications, Oxford University Press.

[57] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[58] T. Horváth. Experiments in Partial Deduction. Master's thesis, Departement Computerwetenschappen, K.U. Leuven, Belgium, 1993.

[59] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, New York, 1993.

[60] Neil D. Jones. Special Issue on Partial Evaluation. *Journal of Functional Programming*, 3(3):251–387, 1993.

[61] Neil D. Jones. An Introduction to Partial Evaluation. *ACM Computing Surveys*, Vol. 28(No. 3), September 1996.

[62] J. Komorowski. An Introduction to Partial Deduction. In A. Pettorossi, editor, *Meta92 Workshop on Meta Programming in Logic*, Lecture Notes in Computer Science 649, pages 49–69. Springer-Verlag, 1992.

[63] Eleftherios Koutsofios and Stephen C. North Drawing graphs with *dot*. AT&T Bell Laboratories, October 1993. Revised version available at `http://www.graphviz.org/Documentation/dotguide.pdf`.

[64] R. Kowalski and D. Kuehner. Linear Resolution with Selection Function. *Artificial Intelligence*, 2:227–260, 1971.

[65] R. A. Kowalski. Predicate Logic as a Programming Language. In Jack L. Rosenfeld, editor,*Proceedings of the Sixth IFIP Congress (Information Processing 74)*, pages 569–574, North-Holland, 1974.

[66] R. A. Kowalski. Logic as a computer language. In *Proceedings Infotec State of the Art Conference, Software Development: Management*, June 1980.

[67] J.B. Kruskal. Well-Quasi-Ordering, the Tree Theorem, and Vazsonyi's Conjecture. *Transactions of the American Mathematical Society*, 95:210–225, 1960.

[68] D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimization. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL'81)*, pages 207–218. ACM Press, 1981.

[69] J. Lam and Kusalik A. A Comparative Analysis of Partial Deductors for Pure Prolog. Technical report, Department of Computational Science, University of Saskatchewan, Canada, May 1991. Revised April 1991.

[70] A.H. Land and A.G. Doig. An Automatic Method for Solving Discrete Programming Problems. *Econometrica*, 28:497–520, 1960.

[71] M. Leuschel and M. Bruynooghe. Logic Program Specialisation Through Partial Deduction: Control Issues. *Theory and Practice of Logic Programming*, 2(4 & 5):461–515, July & September 2002.

[72] M. Leuschel and S. Gruner. Abstract Conjunctive Partial Deduction Using Regular Types and its Application to Model Checking. In *Proceedings of the 11th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'01)*, number 2372 in *Lecture Notes in Computer Science*. Springer, 2001.

[73] M. Leuschel and B. Martens. Partial Deduction of the Ground Representation and its Application to Integrity Checking. In John Lloyd, editor, *Proceedings of ILPS'95, the International Logic Programming Symposium*, Portland, USA, December 1995. MIT Press. Extended version as Technical Report CW 210, K.U. Leuven.

[74] M. Leuschel and B. Martens. Global Control for Partial Deduction Through Characteristic Atoms and Global Trees. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, *Lecture Notes in Computer Science* 1110, pages 263–283, Schloß Dagstuhl, 1996.

[75] M. Leuschel and De Schreye. Logic Program Specialisation: How to be More Specific. In *Proceedings of the International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'96)*, *Lecture Notes in Computer Science* 1140, pages 137–151, 1996.

[76] M. Leuschel and M.H. Sørensen. Redundant Argument Filtering of Logic Programs. In *Proceedings of the 6th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'96)*, pages 83–103. *Lecture Notes in Computer Science* 1207 83–103, 1996.

[77] M. Leuschel and G. Vidal. Forward Slicing by Conjunctive Partial Deduction and Argument Filtering. In *Proceedings of the European Symposium on Programming (ESOP'05)*, Lecture Notes in Computer Science 3444, pages 61–76. Springer-Verlag, Berlin, 2005.

[78] Michael Leuschel. Ecological Partial Deduction: Preserving Characteristic Trees Without Constraints. In Maurizio Proietti, editor, *Proceedings of the 5th International Symposium on Logic-based Program Synthesis and*

Transformation (LOPSTR'95), *Lecture Notes in Computer Science* 1048, pages 1–16, Utrecht, The Netherlands, September 1995. Springer-Verlag.

[79] Michael Leuschel. The ECCE Partial Deduction System and the DPPD Library of Benchmarks. Obtainable via `http://www.ecs.soton.ac.uk/~mal`, 1996-2002.

[80] Michael Leuschel. *Advanced Techniques for Logic Program Specialisation*. PhD thesis, K.U. Leuven, May 1997.

[81] Michael Leuschel. On the Power of Homeomorphic Embedding for Online Termination. In Giorgio Levi, editor, *Proceedings of the International Static Analysis Symposium (SAS'98)*, *Lecture Notes in Computer Science* 1503, pages 230–245, Pisa, Italy, September 1998. Springer-Verlag.

[82] Michael Leuschel and Danny De Schreye. Towards Creating Specialised Integrity Checks Through Partial Evaluation of Meta-Interpreters. In *Proceedings of the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'95)*, pages 253–263, La Jolla, California, June 1995. ACM Press.

[83] Michael Leuschel and Danny De Schreye. Constrained Partial Deduction and the Preservation of Characteristic Trees. *New Generation Computing*, 16:283–342, 1998.

[84] Michael Leuschel, Bern Martens, and Danny De Schreye. Controlling Generalisation and Polyvariance in Partial Deduction of Normal Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208–258, January 1998.

[85] J. W. Lloyd and J. C. Shepherdson. Partial Evaluation in Logic Programming. *The Journal of Logic Programming*, 11:217–242, 1991.

[86] J.W. Lloyd. *Foundations of Logic Programming*. Springer, second, extended edition, 1987.

[87] F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In P. Narendran and M. Rusinowitch, editors, *Proceedings of the 10th International Conference on Rewriting Techniques and*

Applications (RTA'99), pages 244–247. Springer *Lecture Notes in Computer Science* 1631, 1999.

[88] B. Martens and D. De Schreye. Automatic Finite Unfolding Using Well-Founded Measures. *Journal of Logic Programming*, 28(2):89–146, 1996. Abridged and revised version of Technical Report CW180, Departement Computerwetenschappen, K.U.Leuven, October 1993.

[89] John McCarthy, M. L. Minsky, N. Rochester, and C. E. Shannon. A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence. Report, manuscript, MITAI, Cambridge, MA, August 1955.

[90] E. Mera. Development of a prolog profiler. Technical Report CLIP13/2004.1, Technical University of Madrid, School of Computer Science, UPM, September 2004.

[91] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992.

[92] Claudio Ochoa and Germán Puebla. A Study on the Practicality of Poly-Controlled Partial Evaluation. In *Proceedings of the 15th Workshop on Functional and (Constraint) Logic Programming(WFLP 2006)*, pages 123–136. Madrid, Spain, November 2006.

[93] Claudio Ochoa and Germán Puebla. Poly-Controlled Partial Evaluation in Practice. In *Proceedings of the ACM Symposium on Partial Evaluation and Program Manipulation (PEPM'07)*, pages 164–173. Nice, France, January 2007. ACM Press.

[94] C. Ochoa, G. Puebla, and M. Hermenegildo. Removing Superfluous Versions in Polyvariant Specialization of Prolog Programs. In *Proceedings of the 15th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, pages 80–97, number 3901 in *Lecture Notes in Computer Science*. London, UK, April 2006. Springer-Verlag.

[95] C. Ochoa, J. Silva, and G. Vidal. A Lightweight Approach to Program Specialization. In *Proceedings of the IV Jornadas de Programación y Lenguajes (PROLE'04)*, pages 41–54. Málaga, Spain, 2004.

[96] C. Ochoa, J. Silva, and G. Vidal. Dynamic Slicing Based on Redex Trails. In *Proceedings of the ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation (PEPM'04)*, pages 123–134. Verona, Italy, 2004. ACM Press.

[97] C. Ochoa, J. Silva, and G. Vidal. Program Specialization Based on Dynamic Slicing. In *Proceedings of Workshop on Software Analysis and Development for Pervasive Systems (SONDA'04)*, pages 20–31. Verona, Italy, 2004.

[98] Claudio Ochoa and Germán Puebla. A Study on the Practicality of Poly-Controlled Partial Evaluation. In *Proceedings of the 15th Workshop on Functional and (Constraint) Logic Programming (WFLP'06)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2007. To appear.

[99] C. Ochoa, J. Silva, and G. Vidal. Lightweight Program Specialization via Dynamic Slicing. In *Proceedings of the Workshop on Curry and Functional Logic Programming (WCFLP 2005)*, pages 1–7. ACM Press, 2005.

[100] C. Ochoa, J. Silva, and G. Vidal. A Slicing Tool for Lazy Functional Logic Programs. In *Proceedings of the 10th European Conference on Logics in Artificial Intelligence (JELIA'06)*, pages 498–501. Springer *Lecture Notes in Computer Science* 4160, 2006.

[101] Alberto Pettorossi and Maurizio Proietti. Transformation of Logic Programs: Foundations and Techniques. *Journal of Logic Programming*, 19/20:261–320, 1994.

[102] Alberto Pettorossi, Maurizio Proietti, and Sophie Renault. Enhancing Partial Deduction by Unfold/Fold Rules. In John Gallagher, editor, *Proceedings of the 6th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'96)*, *Lecture Notes in Computer Science* 1207, pages 146–168, Stockholm, Sweden, August 1996. Springer-Verlag.

[103] Alberto Pettorossi, Maurizio Proietti, and Sophie Renault. Reducing Nondeterminism While Specializing Logic Programs. In Neil D. Jones, editor, *Proceedings of ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 414–427, Paris, France, January 1997.

[104] Steven Prestwich. The PADDY Partial Deduction System. Technical Report ECRC-92-6, ECRC, Munich, Germany, 1992.

[105] Maurizio Proietti and Alberto Pettorossi. Unfolding — Definition — Folding, in this Order, for Avoiding Unnecessary Variables in Logic Programs. In J. Maluszyński and M. Wirsing, editors, *Proceedings of the International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'91)*, pages 347–g358, Passau, Germany, August 1991. Springer-Verlag, *Lecture Notes in Computer Science* 528.

[106] G. Puebla, E. Albert, and M. Hermenegildo. Efficient Local Unfolding with Ancestor Stacks for Full Prolog. In *Proceedings of the 14th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'04)*, number 3573 in *Lecture Notes in Computer Science*, pages 149–165. Springer-Verlag, 2005.

[107] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in *Lecture Notes in Computer Science*, pages 23–61. Springer-Verlag, September 2000.

[108] G. Puebla and M. Hermenegildo. Implementation of Multiple Specialization in Logic Programs. In *Proceedings ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation (PEPM'95)*, pages 77–87. ACM Press, June 1995.

[109] G. Puebla and M. Hermenegildo. Abstract Multiple Specialization and its Application to Program Parallelization. *Journal of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs*, 41(2&3):279–316, November 1999.

[110] Germán Puebla and Claudio Ochoa. Poly-Controlled Partial Evaluation. In *Proceedings of the 8th ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP'06)*. ACM Press, July 2006.

[111] T. Reps and T. Turnidge. Program Specialization via Program Slicing. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation. Dagstuhl Castle, Germany, February 1996*, pages 409–429. Springer *Lecture Notes in Computer Science* 1110, 1996.

[112] J. A. Robinson. A Machine Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(23):23–41, January 1965.

[113] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[114] D. Sahlin. The Mixtus Approach to the Automatic Evaluation of Full Prolog. In *Proceedings of the North American Conference on Logic Programming*, pages 377–398. MIT Press, October 1990.

[115] D. Sahlin. Mixtus: An Automatic Partial Evaluator for Full Prolog. *New Generation Computing*, 12(1):7–51, 1993.

[116] S. Schoenig and M. Ducasse. A Backward Slicing Algorithm for Prolog. In *Proceedings of the International Static Analysis Symposium (SAS'96)*, pages 317–331. Springer *Lecture Notes in Computer Science* 1145, 1996.

[117] M.H. Sørensen and R. Glück. An Algorithm of Generalization in Positive Supercompilation. In *Proceedings of the International Logic Programming Symposium (ILPS'95)*, pages 465–479. The MIT Press, 1995.

[118] L. Sterling and E. Shapiro *The Art of Prolog.* MIT Press, 1986.

[119] G. Szilagyi, T. Gyimothy, and J. Maluszynski. Static and Dynamic Slicing of Constraint Logic Programs. *Journal Automated Software Engineering*, 9(1):41–65, 2002.

[120] Frank Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[121] Raf Venken and Bart Demoen. A Partial Evaluation System for Prolog: Theoretical and Practical Considerations. *New Generation Computing*, 6(2 & 3):279–290, 1988.

[122] Germán Vidal. Forward Slicing of Multi-Paradigm Declarative Programs Based on Partial Evaluation. In *Proceedings of the 12th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2002)*, pages 219–237. Springer *Lecture Notes in Computer Science* 2664, 2003.

[123] Philip Wadler. Deforestation: Transforming Programs to Eliminate Trees. In H. Ganzinger, editor, *Proceedings of European Symposium on Programming (ESOP'88)*, volume 300 of *Lecture Notes in Computer Science*, pages 344–358, Nancy, France, 1988. Springer-Verlag.

[124] P.L. Wadler. Deforestation: Transforming Programs to Eliminate Intermediate Trees. *Theoretical Computer Science*, 73:231–248, 1990.

[125] D.H.D. Warren. Higher-Order Extensions to Prolog: Are They Needed? In J.E. Hayes, Donald Michie, and Y-H. Pao, editors, *Machine Intelligence 10*, pages 441–454. Ellis Horwood Ltd., Chicester, England, 1982.

[126] Daniel Weise, Roland Conybeare, Erik Ruf, and Scott Seligman. Automatic online partial evaluation. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture. Proceedings*, volume 523 of *Lecture Notes in Computer Science*, pages 165–191, Harvard University, 1991. Springer-Verlag.

[127] M.D. Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, The University of Michigan, 1979.

[128] M.D. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[129] Thomas Williams and Colin Kelley gnuplot - An Interactive Plotting Program. Available at `http://www.gnuplot.info/docs/gnuplot.html`

[130] W. Winsborough. Multiple Specialization Using Minimal-Function Graph Semantics. *Journal of Logic Programming*, 13(2 and 3):259–290, July 1992.